UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Heinz Nixdorf Institut und Institut für Informatik
Fachgebiet Softwaretechnik
Zukunftsmeile 1
33102 Paderborn

Analyzing Self-healing Operations in Mechatronic Systems

PhD Thesis

to obtain the degree of

"Doktor der Naturwissenschaften (Dr. rer. nat.)"

by

Claudia Priesterjahn

Referee:
Prof. Dr. Wilhelm Schäfer

Paderborn, August 28, 2013

Abstract

Self-healing may be used to reduce occurrence probabilities of hazards in mechatronic systems, which are applied in safety-critical environments. Self-healing mechatronic systems may react to failures by a structural reconfiguration of the architecture during runtime. This means, the exchange of components or the modification of the components' connections, in order to avoid that a failure results in a hazard. This reaction is subject to hard real-time constraints because reacting too late does not yield the intended self-healing effects. In order to judge whether a self-healing operation reduces the probability of the hazard successfully, we must consider the failure propagation times and the effect of the self-healing operation on the propagation of failures.

In this thesis, we present an analysis of self-healing operations that particularly considers the two problem mentioned above. This analysis operates on failure propagation models that contain information about propagation times. The failure propagation models are generated automatically from the behavior models of system components.

In the domain of reconfigurable mechatronic systems, not all architectural configurations of a system that are constructed at runtime may be known at design time. This applies in particular to systems of systems. These systems only meet at runtime and establish connections in order to cooperate. This connections may lead to structural configurations that were unknown at design time. However, the analysis of self-healing operations requires concrete architectural configurations. We consequently present a framework that allows for executing our analysis of self-healing operations at runtime. Structural reconfigurations are locked if they construct architectural configurations whose hazard occurrence probabilities exceed the valid hazard occurrence probability of the system. This ensures that architectural configurations that violate the safety requirements of the system cannot be constructed.

Zusammenfassung

Selbstheilung kann in mechatronischen Systemen dazu eingesetzt werden, um die Auftrittswahrscheinlichkeiten von Gefahren zu reduzieren. Selbstheilende mechatronische Systeme reagieren zur Laufzeit auf Fehler im System, indem sie ihre Architektur rekonfigurieren. Das heißt, zur Laufzeit werden Komponenten ausgetauscht oder Kommunikationsverbindungen verändert, um zu vermeiden, dass Fehler Gefahren verursachen. Diese Reaktion unterliegt harten Echtzeitbedingungen, da der beabsichtigte Selbstheilungseffekt bei einer zu späten Reaktion nicht eintritt. Um zu beurteilen, ob eine Selbstheilungsoperation die Auftrittswahrscheinlichkeit einer Gefahr reduziert, müssen die Propagierungszeiten von Fehlern und der Effekt der Selbstheilungsoperation auf die Fehlerpropagierung betrachtet werden.

In dieser Arbeit wird eine Analyse vorgestellt, die Selbstheilungsoperationen analysiert und dabei vor allem die beiden oben genannten Probleme berücksichtigt. Die Analyse wird mit Hilfe von Fehlerpropagierungsmodellen ausgeführt, die Informationen über Propagierungszeiten enthalten. Die Fehlerpropagierungsmodelle werden dabei automatisch aus den Verhaltensmodellen der Systemkomponenten generiert.

Außerdem kann bei rekonfigurierbaren mechatronischen Systemen der Fall auftreten, dass zum Entwurfszeitpunkt nicht alle Systemarchitekturen bekannt sind, die zur Laufzeit erzeugt werden können. Das gilt vor allem für Systeme von Systemen, die sich erst zur Laufzeit treffen und dann zusammen arbeiten. Um Aussagen über den Erfolg von Selbstheilungsaktionen treffen zu können, wird jedoch eine konkrete Systemarchitektur benötigt. Deshalb stellen wir in dieser Arbeit einen Ansatz vor, der es erlaubt, Selbstheilungsoperationen zur Laufzeit zu analysieren. Systemarchitekturen, in denen trotz der Anwendung einer Selbstheilungsoperation die Gefahrenwahrscheinlichkeit den zulässigen Wert übersteigt, werden gesperrt. Damit wird sichergestellt, dass keine Systemarchitekturen erzeugt werden, die die Sicherheitsanfoderungen an das System verletzen.

Acknowledgements

First of all, I thank my supervisor Prof. Dr. Wilhelm Schäfer for the scientific support and the opportunity to work in the inspiring environment of his research group. I thank Prof. Dr. Matthias Tichy and Prof. Dr. Franz-Joseph Rammig for acting as my co-supervisors. Morever, I thank Prof. Dr. Heike Wehrheim and Dr.-Ing. Roman Dumitrescu for attending my exam.

Prof. Dr. Matthias Tichy was a valuable support with lots of discussions about my topics. So was Prof. Dr. Steffen Becker who tought me many soft skills. The current and former PhD-students as well as collegues of our research group Dr. Stefan Henkler, Dr. Martin Hirsch, Dr. Joel Greenyer, Dr. Matthias Meyer, Nicola Danielzik, Ahmet Mehic, Björn Axenath, Dietrich Travkin, Christian Bimmermann, Jan Meyer, Oliver Sudmann, Jörg Holtmann, Stefan Dziwok, Uwe Pohlmann, Tobias Eckardt, Julian Suck, Christopher Brink, Jan Rieke, Matthias Becker, Marie-Christin Platenius, Christian Brenner, and Sebastian Lehrig provided a collegial and trustful environment.

Special thanks go to Markus von Detten and Christian Heinzemann. The first for being a most valuable office colleague. The latter for the excellent cooperation in many scientific topics.

I thank Dominik Steenken, Steffen Ziegert, Christoph Sondermann-Wölke, Jens Geisler, Christian Hölscher, Mareen Vaßholz, Peter Reinold, Kathrin Flaßkamp, Katharina Stahl, and Andry Tanoto for a very good collaboration in the Special Research Center 614.

Special thanks goes to Jutta Haupt and Jürgen Maniera for excellent administrative and technical support.

Moreover, I thank the many students without whom this work would not have been possible. In particular Denis Braun who has been my student worker for several years and Anas Anis and Sebastian Lehrig who where part of an outstanding project group.

I thank Markus von Detten, Matthias Becker, and Steffen Priesterjahn for proof reading.

I thank my parents for providing me with the support to start a scientific career. Foremost, I thank my husband Steffen and my son Tobias for their strong support, their trust and belief in me, and the many times they made the sun shine in my life.

Contents

1 Introduction

The number of technical systems in our world is growing rapidly [Pal08]. A large proportion of these systems are mechatronic systems. Mechatronic systems are developed in a joint effort by mechanical engineers, electrical engineers, control engineers, and software engineers [VDI04]. We find them in our everyday life such as in coffee makers, medical devices, or cars. They are often embedded real-time systems that interact with the real world.

Mechatronic systems are usually software-intensive [SW07]. Their functionality and correctness depend on high quality software. As they are often employed in safety-critical contexts, guaranteeing this high quality becomes an absolute must. Besides testing and simulation, formal verification of a software model and automatic code generation have become an accepted approach to improve software quality and guarantee correctness [Lev95, Sto96].

Formal verification focuses, in particular, on checking safety and liveness properties of the system under development. This prevents the introduction of design faults. However, due to the interaction with the real world, errors in such systems may also be caused by random faults[1]. Random faults may occur, for example, due to the wear of physical components.

Random faults may lead to hazards. Hazards are situations that may lead to accidents which may in turn harm people or property [Lev95]. The goal of the system developer is thus to keep the probability of the occurrence of a hazard acceptable, i.e., below a threshold that has been defined in the system's safety requirements.

If there are hazards in the system whose occurrence probabilities exceed the required threshold, the occurrence probability must be reduced. Of course, it is desirable, to eliminate all hazards from technical systems. This however, is not possible, because in this case most technical systems would not fulfill their function [Lev95]. A reduction of hazard occurrence probabilities may be achieved by self-healing.

Self-healing systems react to observed faults autonomously by executing a self-healing operation that returns the system into a safe state [Sha02]. This self-healing operation may be realized by structural reconfiguration [GSRU07] which is the creation and removal of software components and communication links.

Self-healing offers a cost-efficient way to implement a reliable system, because often failures that cannot be avoided require costly maintenance cycles. However,

[1] According to Avizienis et al. [ALRL04], an error is the deviation from a correct system state. A fault is the cause of an error.

they may be detected and removed automatically at runtime by self-healing operations [GMP+10].

Of course, the developer needs to analyze whether a self-healing operation reduces the occurrence probability of a hazard such that it meets a required threshold. Such an analysis is usually conducted at design time. However at runtime, structural reconfigurations may lead to architectural configurations (from now on configuration) which have been unknown at design time. This applies in particular to systems of systems where several systems are combined at runtime to achieve an extended functionality [SBT11]. Consequently, the effect of self-healing operations on configurations that are only constructed at runtime cannot be analyzed at design time. Still, the developer must guarantee acceptable hazard occurrence probabilities for these configurations. Therefore, self-healing operations must also be analyzed at runtime.

Existing methods that analyze hazards in reconfigurable systems are the approaches of Güdemann et al. [GOR06], Giese et al. [GT06], and Walter et al [WGR+09]. The approach of Güdemann et al. [GOR06] allows for analyzing whether the different configurations of a system lead to hazards. The approach of Giese et al. [GT06] enables the computation of configurations with the lowest and highest hazard occurrence probability. However, both approaches do not take the propagation times of failures into account. They can therefore not analyze how self-healing operations affect failures which propagate through the system. Walter et al.[WGR+09] compute the probability that a system fails despite the fact that self-healing operations are executed. However, they do not analyze how the self-healing operations affect the system.

Existing approaches for runtime analysis do not analyze hazard occurrence probabilities or self-healing operations. They aim at runtime certification [SBT11] or focus on the detection of anomalies in the executed system behavior and try to lead the system back to its intended behavior [DDK+07, FGT11, GMS12, KMM07, Rus08, SRA04].

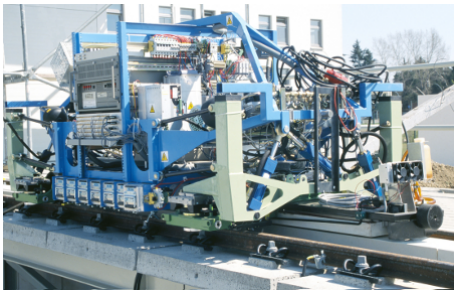
1.1 RailCab

A concrete example of a self-healing mechatronic system is the RailCab[2]. RailCabs are smart rail vehicles that drive autonomously. The vehicles apply the linear drive technology as used by the Transrapid[3] system. Figure 1.1(a) shows a RailCab on the track. Figure 1.1(b) shows the frame of the RailCab and its hardware.

RailCabs drive in a convoy in order to reduce energy consumption caused by air resistance and to achieve a higher system throughput. Such convoys are established on demand and require small distances between the RailCabs. However,

[2] http://www-nbp.upb.de
[3] http://www.transrapid.de

(a) RailCab prototype on the track



(b) Frame and hardware of the RailCab

Figure 1.1: RailCab

these small distances make the coordination of the RailCabs a very safety-critical operation because minor inaccuracies in the coordination can already cause a collision.

Software is an integral part of the RailCab, because a wide range of functionalities such as autonomous driving, active steering, and in particular convoy drive can only be realized with the help of software. Software is used to coordinate the vehicles in real-time, to react in the case of an emergency, and to select and change feedback controllers.

1.2 Problem Definition

The small distances between RailCabs in a convoy are controlled by the drive speed of each RailCab. Based on the speed that is measured by sensors, the embedded software of a speed control subsystem determines the required de-/acceleration to adjust the RailCab's drive speed. If at least one speed sensor fails, a wrong speed propagates to the speed controller. This results in a wrong de-/acceleration, which in turn leads to a wrong distance between at least two RailCabs. This wrong distance may lead to harmful accidents like a collision or derailment that may cause severe property damage or even threaten lives.

A self-healing operation may be specified at design time to prevent such a situation. The self-healing operation replaces the faulty sensor-based speed measurement by a GPS-based speed measurement at runtime. However, the self-healing operation only prevents the hazard if it is performed fast enough such that the wrong speed value does not propagate to the speed controller.

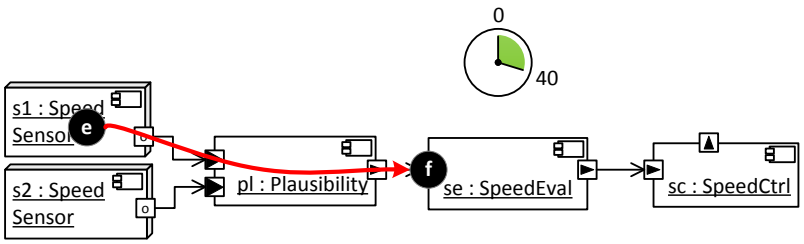
Figures 1.2 and 1.3 show the effects of the self-healing operation for different points in time. Both figures depict the speed control subsystem of the RailCab before and after the application of the self-healing operation.

Before the self-healing operation is applied, the component sc:SpeedCtrl computes the electric current to be set on the linear drive in order to reach a specific speed and thereby keep a specific distance to the RailCab driving in front. The target

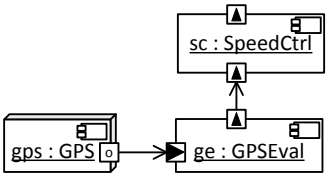
speed is provided by the component se:SpeedEval, which calculates the speed of the RailCab on the track from speed data provided by the speed sensors s1:SpeedSensor and s2:SpeedSensor. If the error in s1:SpeedSensor is detected, a self-healing operation is triggered that removes the link between pos:PosCalc and sc:SpeedCtrl. In Figures 1.2(a) and 1.3(a), a red arrow indicates how far the failures which result from the error in s1:SpeedSensor have propagated at the time when the self-healing operation is executed.

The results after the application of the self-healing operation are depicted in Figures 1.2(b) and 1.3(b). The sensor-based speed measurement consisting of the components s1:SpeedSensor, s2:SpeedSensor, pl:Plausability, and se:SpeedEval have been removed and the components gps:GPS and ge:GPSEval have been created. The speed is now measured based on GPS-data by gps:GPS and evaluated by ge:GPSEval.

In Figure 1.2, the self-healing operation is executed 40 time units after the detection of the error in s1:SpeedSensor. The failure has propagated to se:SpeedEval before the execution of the self-healing operation (cf. Figure 1.2(a)). The self-healing operation removes se:SpeedEval including the failure. The resulting configuration, which is shown in Figure 1.2(b) is thus free from failures. The hazard has been prevented.



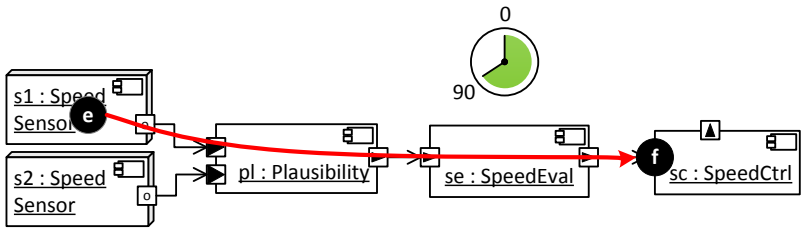
(a) Before applying the self-healing operation



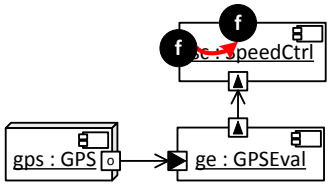
(b) After applying the self-healing operation

Figure 1.2: Self-healing on time

In Figure 1.3, we consider the case that the self-healing operation is executed at 90 time units after error detection. The failure has already propagated to the component sc:SpeedCtrl before the self-healing operation is executed (cf. Figure 1.3(a)). The failure remains in the systems even after the self-healing operation has been applied (cf. Figure 1.3(b)). It will leave sc:SpeedCtrl, propagate to the linear drive and thus cause a wrong speed that leads to a wrong distance between the RailCabs.



(a) Before applying the self-healing operation



(b) After applying the self-healing operation

Figure 1.3: Too late application of a self-healing operation

This example shows that the success of the self-healing operation depends on the location of the failures in the system at the point in time when the self-healing operation is executed. If the failures have propagated too far before the self-healing operation is completed, the self-healing operation fails. To compute the locations of failures at specific points in time, we need to take into account the propagation times of failures.

In order to analyze how self-healing operations affect the propagation of failures, all possible configurations of the system must be known [GT06, PWP+11]. However, in reconfigurable systems, it is possible that configurations occur only at runtime and are unknown at design time [SBT11].

When, for example, RailCabs have become ready for the market, more than one manufacturer will produce them. Then, it will be possible that two vehicles of different manufacturers meet. In order to build a convoy, they need to establish a connection. This connection leads to a configuration that was unknown at design time, because the configuration of the unknown vehicle was, of course, unknown to the developers of the RailCab. The effect of self-healing operations on such configurations needs to be analyzed at runtime.

1.3 Contribution

In this thesis, we present an approach for the *analysis of self-healing operations* (AShOp). It analyzes whether a self-healing operation is executed fast enough to reduce the occurrence probability of a hazard such that it becomes acceptable. This analysis is applied at design time and at runtime. At design time, the developer focuses on constructing self-healing operations that reduce the occurrence probabilities of all possible hazards of the system such that they become acceptable. At runtime, the analysis ensures that only such configurations are

constructed that have acceptable hazard occurrence probabilities. AShOp has been implemented as plugin for the FUJABA Real-time Tool Suite [PTH+10]. We used this tool to apply AShOp on the RailCab and evaluated the scalability.

Figure 1.4 illustrates how the presented methods are applied in a development process. A single person or a group of persons, whose task is to assure the safety of the system, conducts all outlined actions. In the remainder of this document, we refer to these persons as *developer*.

The process starts with the *behavior* models and *static and dynamic architecture* models of the system. The static architecture specifies is the initial configuration of the system. The dynamic part is represented by a set of structural reconfigurations that change the configuration of the system at runtime. Note that the self-healing operations are a subset of the dynamic architecture. The behavior models specify the real-time behavior of the system.

Before the self-healing operations are analyzed, the input models for the analysis need to be created. AShOp uses failure propagation models (*FPM*) [GTS04] with timing annotations. They are generated from the real-time behavior of the system.

Thereafter, the *occurrence probabilities and minimal cut sets*[4](MCS) of the system's hazards are computed. Based on the occurrence probabilities, the developer decides, which hazards have to be reduced by self-healing. The developer constructs self-healing operations for each of these hazards.

AShOp checks whether these self-healing operations guarantee to reduce the probability of a hazard. AShOp requires the static and dynamic architecture, the real-time behavior, the occurrence probabilities and minimal cut sets, and the generated FPMs as input. The result is a *verdict* about the success of the analyzed self-healing operation. It is successful if the occurrence probability of the hazard satisfies the safety requirements after the self-healing.

To analyze self-healing operations at runtime, the reachable configurations are computed at runtime, as well. Then, the self-healing operations are analyzed. Depending on the resulting hazard occurrence probability, the structural reconfigurations, which lead to reachable architecture configurations, are approved or locked.

1.4 Integration into the Development Process

The VDI-Guideline 2206 – "Design methodology for mechatronic systems" [VDI04] provides a guideline for the systematic development of mechatronic systems. The generic procedure is defined by the V-model as illustrated in Figure 1.5.

[4] The combinations of basic events that lead to a hazard [Lev95]

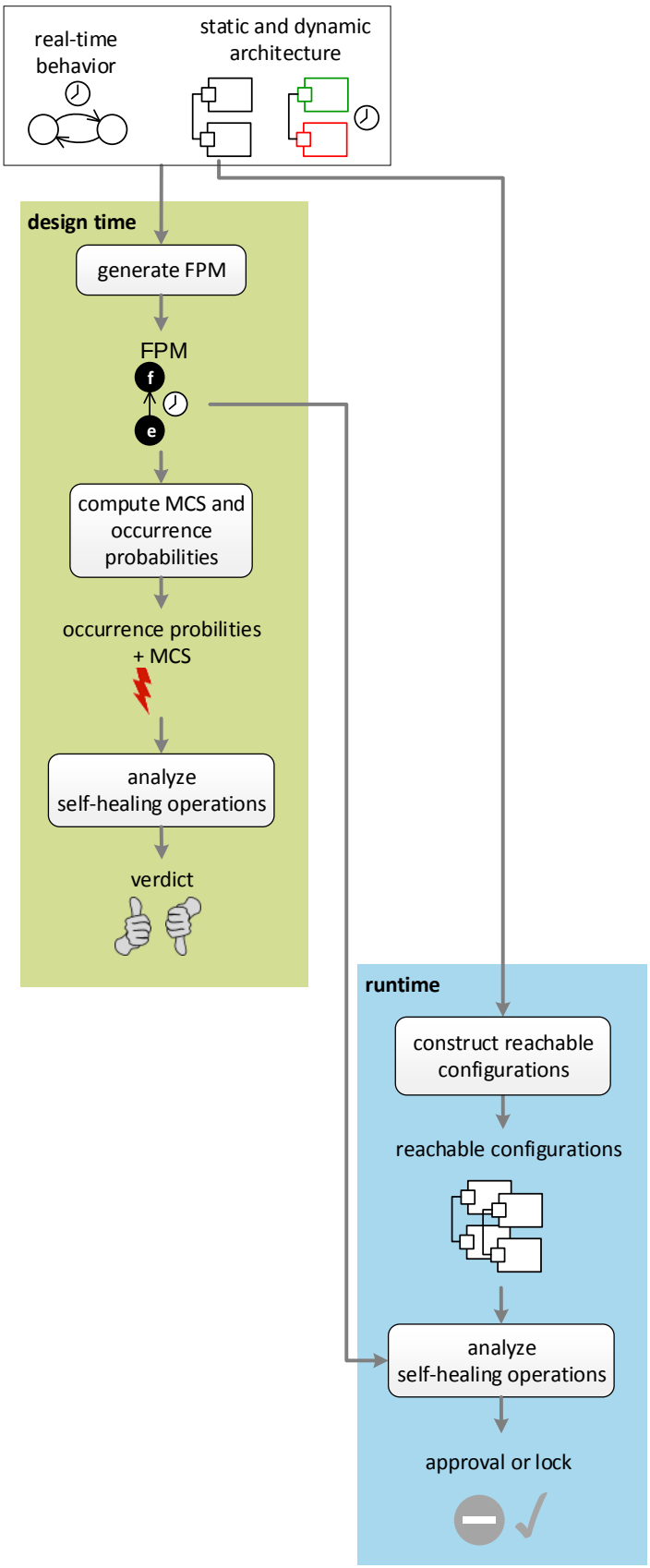


Figure 1.4: Process overview

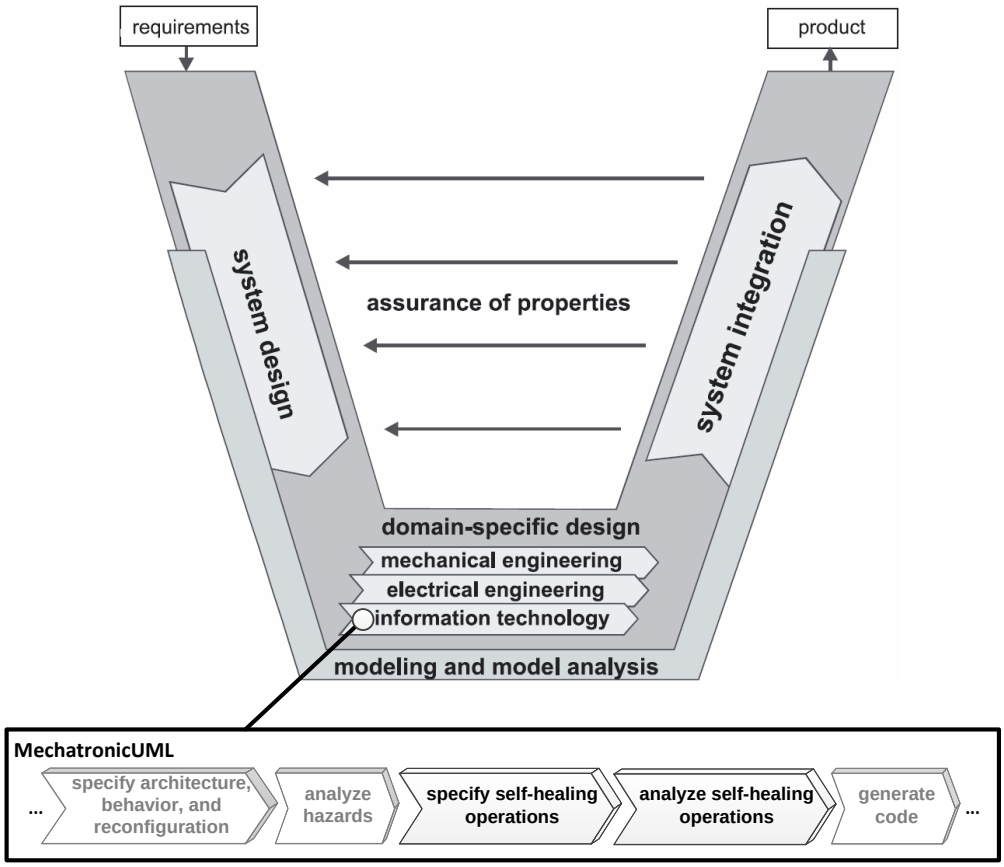


Figure 1.5: V-model for the development of mechatronic systems [VDI04] extended by AShOp

The process starts with the system requirements and ends with the final product. In between, there are three phases: system design, domain-specific design, and system integration. Several cycles of these phases are required for developing a complex mechatronic system.

A high-level solution is constructed during *system design*. This solution covers all domains, which are involved in the system. It specifies the essential physical and logical modes of action. The *domain-specific design* is a concretization of the high-level solution of the system design. During domain-specific design, the high-level solution is extended in each domain separately. Software engineering is embedded in the domain *information technology*. The software architecture is derived from the high-level solution, further refined, and the behavior is specified. During *system integration* the results from the domains are integrated into one system. The interactions of the domain-specific parts are analyzed.

AShOp can only be applied when the architecture and behavior of the system have been modeled. Consequently, it is not carried out during system design. During system integration, the system is assembled and the interaction of the

different system components is tested. AShOp is not applied during this phase either. Consequently, AShOp is only applied during domain-specific-design.

The domain-specific-design is divided into the different domains, which are involved in the development of mechatronic systems. In this thesis, MechatronicUML [BBD+12, EHH+13] is used as a modeling language in the domain of software engineering. A development process has been specified for MechatronicUML in [HSST13] that embeds AShOp as shown in the lower part of Figure 1.5. AShOp is applied when the behavior and structural reconfigurations have been modeled and verified. Then, failure propagation models are generated and the occurrence probabilities of the hazards are computed. Self-healing operations are specified for hazard with too high occurrence probabilities and the effect of each self-healing operation is analyzed by AShOp.

1.5 Thesis Overview

The next chapter presents the foundations needed to introduce the concepts, which result from this thesis. The foundations comprise self-healing mechatronic systems and model-based development techniques for self-healing mechatronic systems. The model-based development includes safety related analysis techniques. In Chapter 3, we introduce our failure propagation models with timing extensions. These failure propagation models are based on a formal semantics. We therefore present a formal definition of all system models first to formalize our new failure propagation models. Chapter 4 continues with the automatic generation of the failure propagation models used in AShOp. The failure propagation models are generated from the behavior models of the system. In Chapter 5, we describe our approach for the hazard analysis of the self-healing behavior (AShOp). Chapter 6 introduces the necessary extensions to analyze self-healing operations at runtime. Chapter 7 presents the tools that implement the concepts described in Chapters 3 to 6. The tools are evaluated with respect to scalability. Chapter 8 discusses works related to the topics of this thesis. We conclude in Chapter 9 with a summary and an outlook on future work.

2 Foundations

In this chapter, we introduce the basic concepts, which are needed to understand the methods that we have developed in the course of this thesis. Therefore, we first define self-healing mechatronic systems - the domain of systems which we consider (cf. Section 2.1). In Section 2.2, we present MechatronicUML. MechatronicUML is a graphical modeling language for the development of software for mechatronic systems. In Section 2.3, we give an overview of the concept of safety and potential threats to safety. In Sec 2.4, we present methods for analyzing hazards.

2.1 Self-healing Mechatronic Systems

Self-healing mechatronic systems are mechatronic systems with self-healing capabilities. Self-healing means that the systems monitors itself at runtime to detect errors. Once, an error was detected, the system autonomously prevents that the error causes malfunctions or even hazards.

2.1.1 Mechatronic Systems

Mechatronic systems, which have to be developed in a joint effort by teams of mechanical engineers, electrical engineers, control engineers, and software engineers, enable innovative design concepts. Each mechatronic system comprises four components as depicted in Figure 2.1. The *basic system* is a mechanical, electromechanical, hydraulic, or pneumatic structure or a combination of these. The state of the basic system is measured by different types of suitable *sensors*. The *information processing* unit contains control algorithms, which calculate the desired values for the actuators. The *actuators* influence the behavior of the system. Additionally, three types of flow are necessary to describe the interaction between the four basic components. The *information flow* is used to exchange pieces of information like measured values. The *energy flow* describes the type and amount of energy transferred mainly from and to the basic system. The *material flow* comes into play, where raw material or semi-finished products are used to fabricate the desired product, or where inherent processes use material like oil for hydraulic components.

The information flow between the sensors, the basic system, the actuators, and the information processing builds a feedback loop: The actuators affect the basic system. The behavior of the basic system is measured by the sensors,

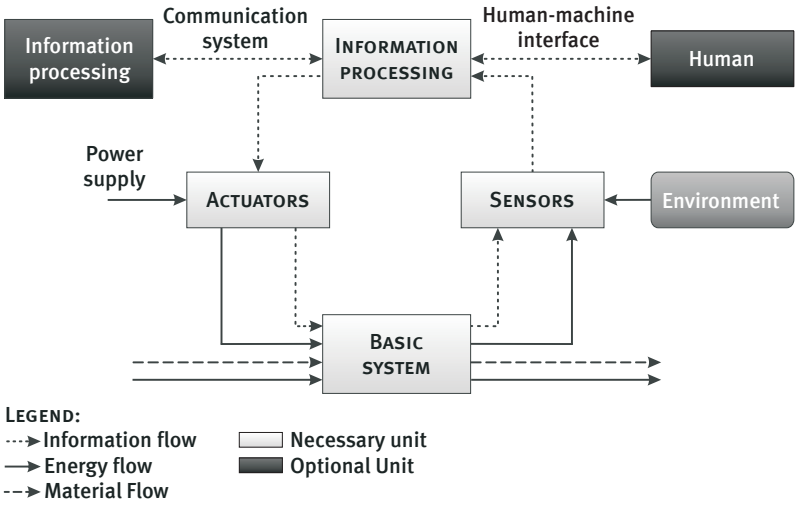


Figure 2.1: Basic structure of a mechatronic system [VDI04]

which forward this information to the information processing. The information processing, in turn, controls the actuators based on the information of the sensors.

The actuators of mechatronic systems interact with the physical world. Consequently, they have to operate in real-time. The behavior of *real-time systems* not only depends on the results of computations but also on the point in time when a result is delivered. The point of time when a result is needed is specified by *deadlines*. Depending on the consequences in cases where results are delivered too late, real-time systems are divided into soft and hard real-time systems. In *hard real-time* systems, the miss of a deadline leads to a system failure. In *soft real-time* systems, the results may still be used but its usefulness may be reduced [Kop97].

A key property of software, which is applied in mechatronic systems, is its exposure to hardware failures. Hardware failures differ from software failures in that hardware fails randomly. The correct functionality of sensors as an example of hardware components is essential for the safe operation of mechatronic systems. Sensors provide information about the behavior of the actuators and the system in its environment. Broken sensors need to be replaced immediately by working sensors to prevent accidents. Therefore, we particularly concentrate on sensor failures and their recovery.

2.1.2 Self-healing Systems

All parts of a mechatronic system, either hardware or software, may fail and thus affect its behavior. This may lead to dangerous situations, namely *hazards*, where people or the environment may be harmed. *Self-healing* may be used to prevent hazards in the case of failure. For the understanding of self-healing, we

clarify first specify a healthy system. Shaw [Sha02] defines a healthy system as follows:

> "A system is considered healthy as long as it works according to its specification. As a system may be used in many ways, criteria for system health depend on the respective application of the system" [Sha02].

In other words, a system is healthy as long as it shows its intended behavior. Self-healing comes into play if the system is not healthy, i.e., it deviates from its intended behavior.

As proposed by Carzaniga et al. [CGP08], we refer to self-healing by the ability of a system to autonomously detect and overcome failures. Self-healing aims at correcting a system in the case of failure such that failures can be tolerated. Thus, self-healing is closely related to fault tolerance.

Fault tolerance is the characteristic of a system that faults do not result in a system failure. Fault tolerance is realized by employing some kind of redundancy [Sto96]. If parts of the basic system fail, backup components continue the operation of the basic computer system. If one component fails, the remaining components still provide the functionality (static redundancy) or broken components are replaced by spares (dynamic redundancy). Dynamic redundancy is referred to as self-healing [Sto96].

One approach for the self-healing of mechatronic systems are *artificial immune systems* [MR12, RSV13]. Artificial immune systems adapt the principles of biological immune systems: If a cell of an organism is infected by a pathogen, special molecules of that cell take the signature of the infecting pathogen. When the signature is detected, the immune systems starts producing antibodies that kill the infected cell to heal the organism. The biological immune system is also able to adapt to different kinds of pathogens. These principles are transfered to artificial immune systems by implementing the cells of the biological immune system as software agents[1] [MR12].

In artificial immune systems, the behavior of the system is not given in advance. Instead, the artificial immune systems learns the behavior of the system by observation at runtime. The borders between normal and anomalous behavior are specified at the same time. Anomalous system behavior is then detected by observing the system behavior. The system is returned to normal behavior by a immune response that switches off the broken system part in the same manner that a biological immune system kills infected cells. In a computer system, this may be switching off broken parts of an FPGA[2] [MR12].

However, the immune system does not analyze the system behavior that results from the immune response. Artificial immune systems can therefore only be

[1] A software agent is a program that acts autonomously and amongst other capabilities is able to learn from observed behavior[WJ95].

[2] Field Programmable Gate Array [Rea11]

applied in mechatronic systems, if there is a subsystem that analyzes the effect and timing of the immune responses and controls them.

2.1.3 Self-healing Process

Self-healing requires a series of actions called *self-healing process*. According to Gorla et al. [GMP+10] self-healing comprises the following steps: failure detection, fault identification and execution of the self-healing operation. In this work, we focus on the execution of the self-healing operation.

Failure Detection

When for example a sensor is broken, it generates a signal, which deviates from the correct signal. To detect a broken sensor, the designer needs to know the correct signal in advance. Correct sensor output can be predicted in most cases, because for most safety-critical systems the behavior is known in advance, e.g., from simulation: Every time, a control command is issued in a mechatronic system, an actuator alters the system. The response measured by the sensors will match the command. Otherwise something has failed. For error detection, the software may, for example, compare actual sensor responses with issued commands [Dun02], perform self-checks by, e.g. triggering a known value at the sensor [Mes01], or the system may contain self-validating sensors [HC93, Cla95, Ise07]. In artificial immune systems, failures are detected by observing the system behavior and comparing it with the learned behavior. Thereby, it is difficult to decide whether a deviating behavior is already a failure or still normal [RSV13]. It is, of course, essential to detect failures as early as possible to leave the maximum time for recovery.

Fault Identification

When a failure has been detected the causing fault needs to be located and its type to be determined [LKN+11]. Only then, it can be prevented from causing further failures. The detected failure may occur anywhere in the system, far away from the causing fault. In the case of a sensor fault for example, the failure may be detected at an actuator. The actuator and the sensor may be separated by several software components that process the sensor signal and output control commands.

In order to pick an appropriate self-healing operation, the fault needs to be identified. However, the relation between fault and failure mostly is not one-to-one. Faults may result in several failures and a failure can be caused by many faults. The effect of a fault may only result in a failure after long executions [GMP+10].

There exist many techniques to identify faults, for example [BBI+11, Joh03, dKMR92, dKW87, MDDW05, Rei87]. Faults may be identified by the comparison of failing executions with behavior models [GMP+10]. Another option is

to use fault dictionaries [PR92] A *fault dictionary* stores the observed system behavior which is caused by a set of faults. This is done by testing the system with fault models. The fault identification compares the observed system behavior with the precomputed responses and looks up the fault in the fault dictionary [AO02]. Fault models are used in both cases. Consequently, faults that have not been modeled cannot be detected. There also exist fault identification techniques that work without fault models, but they require much more effort. Their application in real-time systems is thus not feasible [BO10].

In this work, we use fault dictionaries for fault identification. The advantage of fault dictionaries is their time efficiency which makes them applicable to real-time systems [BO10]. However, if the fault dictionary becomes too large, searching the book may take a long time. Consequently, many approaches focus on an efficient representation of fault dictionaries [BHF96, PR92, RFP93, CKKK06].

In artificial immune systems, the fault itself is not needed to select an appropriate immune response. Instead, the reaction depends on the signature of the failure [MR12, RSV13].

Self-healing Operation

The last step of the self-healing process is the actual self-healing operation, which removes the fault from the system and stops failures from propagating any further. In this work, we particularly focus on self-healing in the form of structural reconfiguration, because a self-healing operation can only return a system to a healthy state by removing failed system parts. On the one hand, the self-healing operation may simply remove failed system components. In this case either existing system components provide the functions of the failed components or the system is degraded. On the other hand, the self-healing operation may replace failed components by creating new software components. In both cases, the structure of the system is changed [GSRU07]. We consequently model these changes by structural reconfiguration.

Remark 2.1 *In this work, we model self-healing operations by structural reconfigurations. This means, the self-healing operations are a subset of the reconfiguration rules of the system.*

Thereby, the concept of encapsulation is preserved: behavior is changed in the sense of removing or adding functions which are encapsulated in system components. This helps the developer in understanding the system model. The comprehensibility of reconfiguration rules, which are based on components is further supported by the opportunity to model hierarchical architectures which allows for specifying system architecture and reconfiguration on different levels of abstraction.

Once a fault has been identified, the failed components are removed from the basic system using structural reconfiguration. This must happen before failures propagate to critical parts in the system, i.e., before they may cause hazards.

In artificial immune systems, the kind of immune response differs depending on the system level it is applied to. On the hardware level, the immune response may switch off hardware that is broken, e.g., parts of an FPGA [MR12]. On operating system level, tasks may be restarted or killed [RSV13].

2.1.4 Integration into the CRC 614

In the Collaborative Research Center 614 "Self-optimizing concepts and structures in mechanical engineering" (CRC 614), we develop design techniques for self-optimizing mechatronic systems. In order to show the integration of this thesis into the CRC 614, we sketch how self-optimization is used to implement self-healing.

Self-optimizing Mechatronic Systems

Self-optimizing mechatronic systems aim at optimizing their behavior. They evaluate the information, which is collected by their sensors and react optimally to changing operation conditions. This is the *paradigm of self-optimization*. It is implemented in the information processing of the mechatronic system (cf. Figure 2.1). In the CRC 614, self-optimization is realized by behavior adaptation which is implemented by parameter adaptation or structural reconfiguration [ADG+09].

Self-optimization may be used to recover a system in the case of failure, i.e., to heal the system. Thus, self-healing may be considered as a special case of self-optimization. According to Adelt et al. [ADG+09], the self-optimization of a technical system is the adaptation of the system objectives to changing influences on the technical system and the resulting autonomous adaptation of the parameters and the structure of the system. The adaptation of the system structure causes an adaptation of the system behavior.

Self-optimizing mechatronic systems determine objectives autonomously and adapt their behavior correspondingly. According to Gausemeier et al. [GFDK09], self-optimization is defined by a series of three actions that define the *self-optimization process*.

1. Analysis of the Current Situation The current situation consists of the system state and current and former observations of the system environment. This information may be collected directly but also indirectly by information exchange with adjacent systems. The analysis of the current situation determines whether the current system objectives are suitable for the current situation. This action corresponds to the first two actions of the self-healing process: failure detection and fault identification.

2. Setting the System Objectives The system objectives are set by selection, adaptation, or generation. A selection is made from a given set of alternative system objectives. Adaptation means the modification of the characteristics or the weights of system objectives. A generation creates new system objectives independently from existing system objectives.

This action has no correspondence in the self-healing process. However, it is possible to specify multiple self-healing operations as a reaction to a failure. In this case, the self-healing operation would have to be chosen according to the system objectives.

3. Adaptation of the System Behavior The adaptation of the system behavior is achieved by modifying system parameters or the system structure. This action represents the final feedback of the self-optimization cycle. In the self-healing process, this action is the self-healing operation.

The self-optimization process may be executed reactively or proactively. Reactive means, the system reacts to changing operational parameters, e.g., changes in the environment. Proactive means, the system acts in terms of planning, e.g., in order to transport goods as energy-efficient as possible. For self-healing, reactive adaptation is applied: The system detects a failure and reacts by an appropriate self-healing operation.

Development of Self-optimizing Mechatronic Systems

The development process for self-optimizing systems is closely related to the development process of the VDI-Guideline 2206 – "Design methodology for mechatronic systems" [VDI04]. The development of self-optimizing mechatronic systems requires the collaboration of the involved disciplines. They need to be organized in order to design and construct the system such that it fulfills its requirements. The disciplines are organized by a domain-spanning system specification, which is called *principle solution* as introduced by Gausemeier et al. [GSG+09]. The principle solution contains all aspects that concern more than one domain. It is constructed with the collaboration of all involved disciplines in the first design phase of the development, which is called *interdisciplinary conceptual design*. The principle solution comprises several languages for specifying different aspects of the system. They describe the requirements, logical and physical structure, shape, and behavior of the system [HSST13].

The next phase *design and development* extends the principle solution in the different domains and integrates the results of the different domains into the final system. Each domain uses its domain-specific languages. The design and development is coordinated by the principle solution [GSG+09]. For the domain of software engineering, an initial architecture and behavior of the software is derived from the principle solution [ADG+09, GGS+07]. The architecture and behavior are extended using the language MechatronicUML, which will be explained in detail in Section 2.2.

Of course, the principle solution and the models of the different domains need to be kept consistent [GGS+07, GSG+09], in order to coordinate the different domains throughout the whole development process. Therefore, changes of the domain-specific models are transferred automatically to the partial models of the principle solution. This synchronization is managed by correspondence models, which define a relation between the principle solution and the domain-specific models. These changes in the principle solution are transferred to other domain-specific models as well, if their correspondence is not fulfilled any more.

Operator-Controller-Module

An increasing amount of functionality of mechatronic systems is implemented by software. Consequently, information processing in such systems becomes more and more complex [GFDK09]. A huge amount of system components needs to be coordinated in order to achieve the pursued system objectives. Software coordinates components that control the system's hardware and optimization algorithms adapt the controlling software to changes in the system environment.

In order to achieve the controllability of the information processing, the different functions must be structured reasonably. Gausemeier et al. [GFDK09] propose the *Operator-Controller-Module* (OCM) as shown in Figure 2.2. The OCM is divided into three levels – the cognitive operator, the reflective operator and the controller.

The *controller* contains the control engineering parts of the information processing. It processes directly the signals measured by the sensors, computes adjustment signals and outputs them to the controlled hardware. The *reflective operator* monitors the controller. It directs the controller by triggering parameter changes and structural reconfiguration. The *cognitive operator* provides knowledge about the system itself and the system's environment. This knowledge is used to optimize the system behavior.

The controller and parts of the reflective operator are executed in hard real-time. The cognitive operator and the other parts of the reflective operator are executed in soft real-time.

The self-healing is embedded in the reflective operator, because it is executed in hard real-time but does not control the hardware directly: If a self-healing operation misses its deadline, a hazard may occur and thus the self-healing operation would not yield its intended effect.

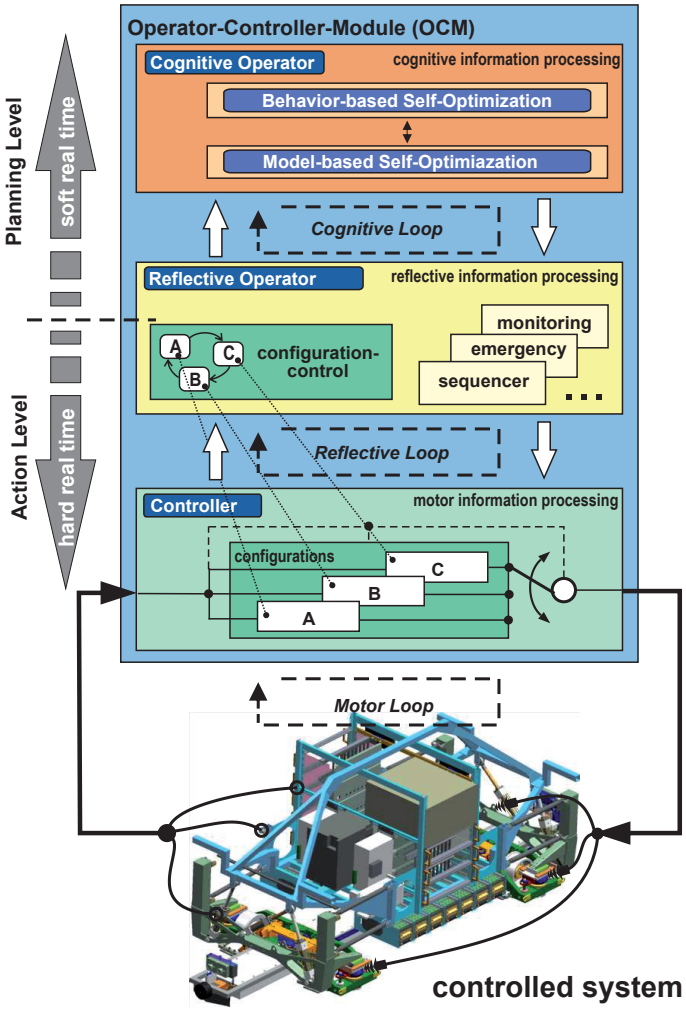


Figure 2.2: Structure of the Operator-Controller-Module [GFDK09]

2.2 MechatronicUML

In this section, we introduce our modeling language MechatronicUML as presented in [BBD+12, BDG+11, EHH+13, HPB12]. MechatronicUML adapts concepts of the Unified Modeling Language (UML) [Gro09] to model the software of mechatronic systems. The MechatronicUML method enables the model-driven design of discrete software of self-adaptive mechatronic systems. The key concepts of MechatronicUML are a component-based system model, which enables scalable compositional verification of safety properties, the model-driven specification and verification of reconfiguration operations, and the integration of the discrete software with the controllers of the mechatronic system. Therefore, MechatronicUML provides a set of domain specific visual languages as well as a defined development process.

We derive our static architecture from UML component and deployment diagrams. For the dynamic part, we consider two types of behavior: reconfiguration and stateful internal behavior / message passing. For reconfigurations, we use extended graph transformation rules that we apply to the graphs that constitute component structures. For modeling the internal behavior of the component instances and their communication with each other, we use real-time statecharts – UML State machines extended by clocks and constraints over these clocks.

The process of MechatronicUML [HSST13] starts with specifying the modules of the component types (cf. Sec 2.2.1) and communication protocols called "coordination patterns" (cf. Section 2.2.1). Then, real-time behavior is added to the component types (cf. Section 2.2.2). An initial system architecture is specified by component instance configurations (cf. Section 2.2.1) for software only and by deployment diagrams (cf. Section 2.2.1) for software which is deployed on a hardware. The structural reconfiguration is specified (cf. Section 2.2.3) based on the initial system architecture and integrated via side effects of the components' real-time behavior.

Remark 2.2 *In the course of this thesis, we will use* MechatronicUML *to model system architectures, reconfigurations, and behavior.*

2.2.1 Component Model

In MechatronicUML, the static system architecture is specified by components. A component encapsulates its inner structure and behavior and allows interaction with other components only via its interfaces. MechatronicUML distinguishes between component types and component instances. A component instance is the occurrence of a component type. Component instances of one component type have the same behavior specification but may be in different system states. The component model allows for multiple instantiation of components within a system.

In MechatronicUML, the interfaces are realized by ports. A component type defines a set of named port types. Further, port types specify a cardinality with

a lower and upper bound. The lower and upper bounds define the minimum and maximum number of possible port instances of a port. Ports can be either single ports or multi-ports. *Single ports* can be connected to only one communication partner. *multi-ports* can be connected to one or more communication partners.

MechatronicUML further distinguishes three kinds of ports depending on the kind of transmitted information: *discrete ports*, *continuous ports*, and *hybrid ports*. A discrete port is used to send and receive discrete, asynchronous data. Continuous ports are used to transmit signals. A "signal is a time varying quantity that has values at all points in time"[3]. Hybrid ports combine discrete ports and continuous ports. This means, a hybrid port can process discrete and continuous data. Hybrid ports are used to connect software components that process discrete data to hardware components that process continuous data.

The MechatronicUML component model further distinguishes between atomic components and structured components. Atomic components are active objects and contain a stateful behavior specification. Structured components are assembled by embedding other components. The embedded component types are called *component parts*. The embedding component type is called *structured component type*. A structured component type does not contain a behavior by itself, because the behavior of the structured component type is defined by the behavior specification of its component parts.

Figure 2.3 shows the atomic component type PosCalc. Component types are represented by rectangles. Port types are drawn as squares containing triangles. The orientation of the triangle indicates the orientation of the communication. The atomic component type PosCalc has the three hybrid port types speed1, speed2, and gps, and the discrete port types sender and modeIn. In contrast to discrete ports, the hybrid port contains a triangle that touches the boarders of the square.

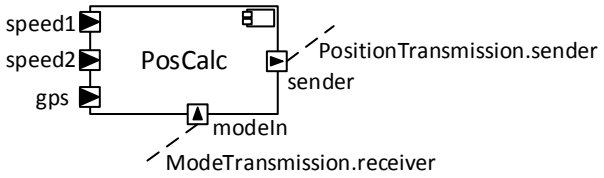


Figure 2.3: Atomic component type PosCalc

Figure 2.4 shows the structured component type RailCab and an excerpt of its component parts. Component parts are represented by rectangles that are contained in a component type. The structured component type RailCab contains the two component parts SpeedControl and Coordination and has one multi-port of the type coordinator. The component part SpeedControl has one single port which is connected to the single port of Coordination by a *connector*. The multi-port coordinator of Coordination is delegated to the multi-port coordinator of RailCab.

[3] `http://www.mathworks.de/help/toolbox/simulink/ug/f15-99132.html`

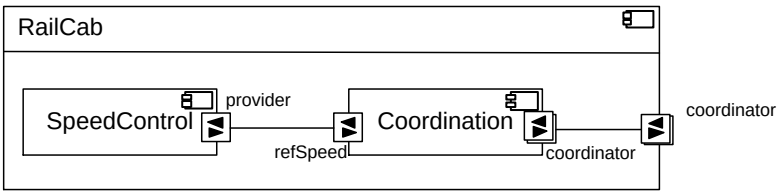


Figure 2.4: Structured component type RailCab

MechatronicUML uses hardware nodes to model hardware entities. Hardware nodes only exist on instance level. They are drawn as boxes as depicted in Figure 2.5. They communicate via hardware ports. Each hardware port contains either an "i" or an "o". "i" means incoming port and "o" means outgoing port. Signals are received via incoming ports and sent via outgoing ports. The hardware node in Figure 2.5 represents a distance sensor that sends signals via two outgoing ports.

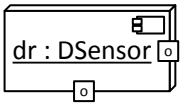


Figure 2.5: Hardware node ds:DSensor

In MechatronicUML, hardware nodes do not have a state-based behavior [BBD+12]. They are used to model the propagation of hardware failures into the software.

Remark 2.3 *In this thesis, we will use the term "component" as a generic term for component type, component instance, and hardware node.*

Coordination Patterns

A *coordination pattern* is used to model the communication protocol between communicating components [GTB+03]. The communicating components are represented by *roles*. A coordination pattern consists of two roles and a connector linking the roles. We distinguish single roles and multi-roles. *Single roles* can communicate with one communication partner only. *Multi-roles* may communicate with more than one communication partner. The role behavior is implemented by ports that build the interfaces of our components. The behavior of roles and ports is specified by real-time statecharts which will be described in Section 2.2.2.

Figure 2.6 shows the concrete syntax of our extensions. *Coordination patterns* are represented by dashed ellipses. Roles are drawn as dashed squares. Bidirectional connectors are illustrated by lines between port types. Unidirectional connectors are drawn as arrows. In this example, the coordination between two RailCabs is modeled by the ConvoyCoordination pattern. It specifies the protocol

to coordinate two RailCabs driving in a convoy. It consists of the multi-role coordinator, the single role member, and one connector that models the link between the two RailCabs. The annotations at the roles specify their cardinalities. The multi-role coordinator can have one to ten communication partners.

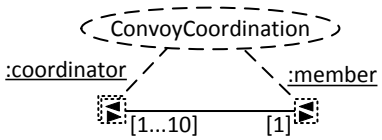


Figure 2.6: Coordination pattern ConvoyCoordination

A multi-role defines the behavior of a role which communicates with several single roles (cf. Figure 2.7). All simple roles have the same behavior. A multi-role is a parallel composition of a set of *subroles* each communicating asynchronously with a simple role. Subroles communicate synchronously with each other. An adaptation behavior initializes the multi-role and also initiates the communication between subroles and creates each subrole.

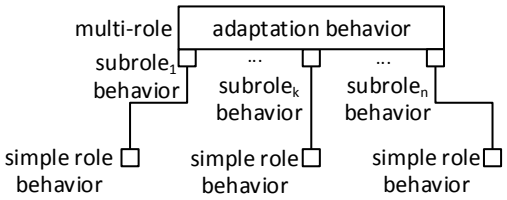


Figure 2.7: Schematic structure of a parameterized coordination pattern

The behavior of a multi-role is specified by the behavior of each subrole and the adaptation behavior. The adaptation behavior triggers the communication of the first subrole. Each subrole triggers the next subrole and the last subrole again triggers the adaptation behavior and so forth.

Component Instance Configuration

Our component model distinguishes between component types and component instances. It allows for multiple instantiations of component types within a system. Note that ports are never instantiated on their own. They are rather instantiated as adjuncts of their respective component type.

A component instance configuration is a concrete assembly of software component instances, connected by connectors at the port instances. We illustrate component instance configurations by UML component diagrams. Figure 2.8 shows a component instance configuration of two RailCabs driving in a convoy. Component and port instances have the same syntax as component and port types. The names of component and port instances are underlined. The name of a component instance has two parts. The unique name of the component

instance is located before the colon, the component type of the component instance is written behind the colon. The names of port instances are prefixed by colons.

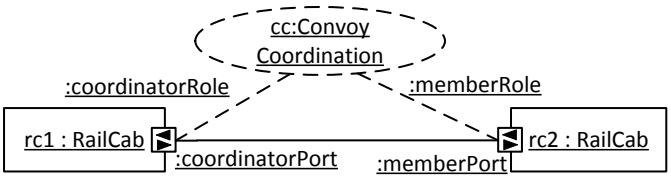


Figure 2.8: Component instance configuration

The instances of coordination patterns have the same syntax as coordination patterns. In Figure 2.8, the RailCab instances communicate via the instance cc of the ConvoyCoordination coordination pattern. Names of coordination pattern instances are underlined as well and have the same layout as names of component instances: unique name and coordination pattern name divided by a colon. The lines which connect the oval of the coordination pattern and the ports carry the underlined names of the roles (e.g. coordinatorRole) that the port implements (e.g. coordinatorPort) prefixed by colons. In the remainder of this thesis, we will omit information about coordination pattern instances and role instances in component instance configurations, because this information is not needed to illustrate the techniques, which are presented in this thesis. Port names will only be annotated if needed.

Deployment Diagram

We use deployment diagrams to model the deployment of software on hardware. A deployment consists of a component instance configuration and a set of hardware nodes.

Of course, each software component is executed on a hardware node, for example an electronic control unit (ECU). This ECU may fail randomly and for example compute wrong values. Like this ECU, each physical entity of a system may fail randomly. To take into account all possible hardware failures of the system, each physical entity must be represented in the deployment diagram [PSTH11]. In this work, however, we focus on the failure propagation over time and not on the completeness of the model. To make our examples in this thesis more comprehensible, we therefore only show physical entities that help understanding our approach.

Figure 2.9 shows a deployment of a component instance of the type RailCab on the hardware node ecu:ECU that represents an ECU. Component instances have the same syntax as in component instance configurations (cf. Section 2.2.1).

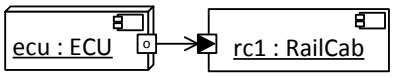
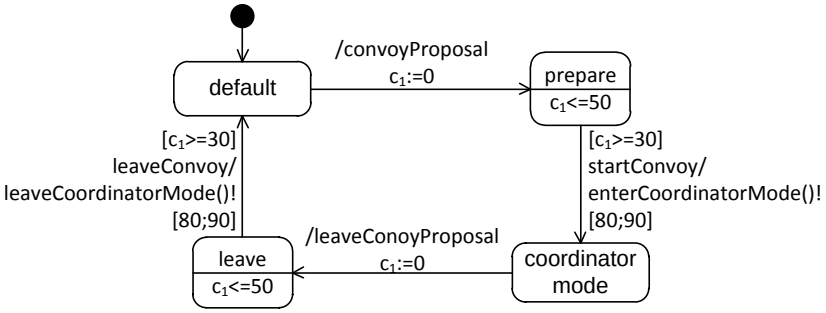


Figure 2.9: Deployment diagram

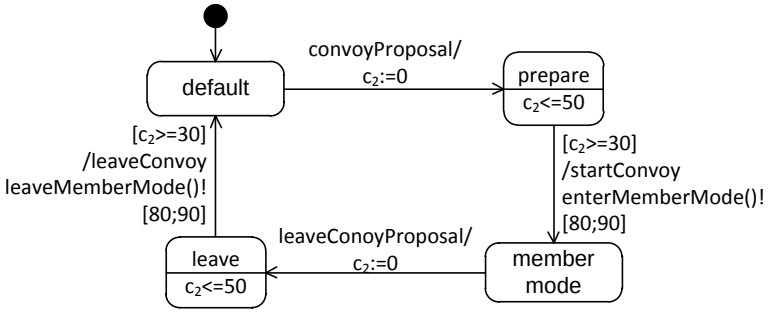
2.2.2 Behavior Models

The behavior of component types, component instances, roles, ports, and connectors is specified by real-time statecharts [GTB+03]. Real-time statecharts are an extension of UML state machines that support more powerful concepts for the specification of real-time behavior. They contain clocks and constraints over these clocks, which enables the specification of time-dependent behavior. *Deadlines* allow for specifying that time passes during the execution of a transition.

Figure 2.10 shows the real-time statecharts of ports of the coordinator role and member role of the ConvoyCoordination coordination pattern. In real-time statecharts, states are represented by rounded rectangles. Transitions are drawn as arrows. The real-time statecharts of Figure 2.10 only show the basic behavior of building a convoy. Most acknowledgments, negotiation, and decisions about reactions to messages are neglected.



(a) Coordinator port



(b) Member port

Figure 2.10: Real-time statecharts of ports of the ConvoyCoordination pattern

The real-time statechart in Figure 2.10(a) has a clock c_1. Based on clocks, real-time statecharts specify time guards, clock resets, and invariants as known from

timed automata [Alu99]. A timed automaton is a finite automaton extended by clocks and constraints over the clocks. It thus allows for modeling real-time behavior.

A time guard is a clock constraint that restricts the execution of a transition to a specific time interval. A clock reset sets the value of a clock back to zero while a transition is fired. Invariants are clock constraints associated with locations that forbid that a timed automaton stays in a location when the clock values exceed the values of the invariant.

The real-time statechart in Figure 2.10(a) has the clock reset $c_1 := 0$ at the transition between default and prepare. The transition from prepare to coordinatormode has the time guard $[c_1 \geq 30]$. The time invariant in state prepare is $c_1 \leq 50$. Both time constraints together limit the firing of the transition to the time when c_1 has a value between 30 and 50.

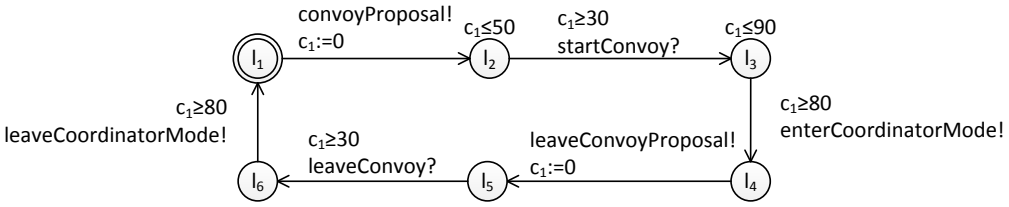
Events are defined by ports and occur as trigger or raised events, which are attached to the transitions of a real-time statechart. A trigger event is placed in front of a "/", a raised event is placed behind it. These events abstract from real data such as sensor values. Besides events, transitions of a real-time statechart may carry synchronizations and side effects. Events model asynchronous communication between real-time statecharts. Synchronizations model synchronous communication between real-time statecharts of the same multi-role and inside one component. They are defined analogously to UPPAAL synchronizations (cf. [BY03]), i.e., they are marked with "?" or "!" for received or initiated synchronizations, respectively. Side effects are methods being executed by the corresponding component when the transition fires.

In Figure 2.10 both ports are initially in state default, which means that they are not running in a convoy. When the coordinator port decides to propose building a convoy, an event convoyProposal is sent to the member port. Both real-time statecharts reset their clocks c_1 and c_2 when the transitions are fired and switch to the state prepare. Next, the member port acknowledges by sending the message startConvoy which is received by the coordinator port. The member port switches into the state member mode and sends the synchronization enterMemberMode()!. This synchronization causes the component internal behavior to create the necessary component for the RailCab to act as a convoy member. The real-time statechart of the coordinator port switches into the state coordinator mode and initiates the reconfiguration for convoy mode by the synchronization enterCoordinatorMode()!. The transitions must fire the earliest at a clock value of $c_1 \geq 30$ and the latest at a clock value of $c_1 \leq 50$ as specified by the time invariant $c_1 \leq 50$ in state prepare and the time guard $c_1 \geq 30$ at the outgoing transition of state prepare. The transition carries the deadline [80,90] which specifies that the transition needs at minimum of 80 and at maximum of 90 time units. Eventually, the coordinator port decides to break up the convoy and sends a corresponding proposal to the member port. The communication and reconfiguration triggers are analog to the communication and reconfiguration triggers needed to build the convoy.

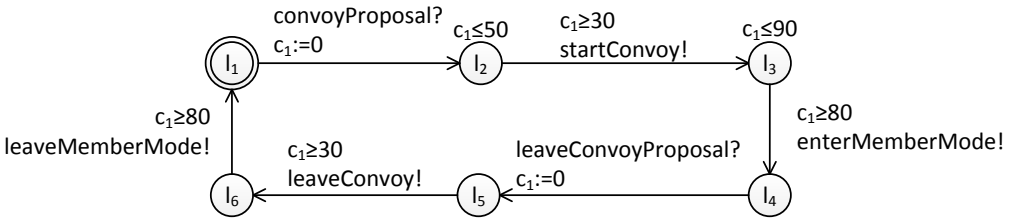
The behavior of a component type or instance is specified by only one real-time statechart. This real-time statechart consists of one state in which all real-time statecharts of the component type or instance are embedded in compartments.

Formally, real-time statecharts are an extension of timed automata. A mapping from real-time statecharts to timed automata was defined in [GB03]. This mapping allows for, e.g., verifying real-time statecharts using the model checker UPPAAL [BDL04].

Figure 2.11 shows the timed automata that correspond to the real-time statecharts in Figure 2.10. Circles represent locations. Arrows depict transitions. Invariants, time guards, and clock reset have the same syntax as in real-time statecharts.



(a) Timed Automaton of the real-time statechart of the coordinator port of Figure 2.10(a)



(b) Timed Automaton of the real-time statechart of the member port of Figure 2.10(b)

Figure 2.11: Timed automata of the real-time statecharts of Figure 2.10

Transitions of real-time statecharts that have a deadline are split into two transitions and a location [GB03], because transitions in timed automata fire in zero time. Time may only pass in locations. Therefore, an extra location is added, where the time of the transition may pass. An example of such a transition in a real-time statechart is the transition from prepare to coordinatiormode in Figure 2.10(a). It has the deadline $[80, 90]$. In the corresponding timed automaton in Figure 2.11(a), this transitions is represented by the transitions from l_2 to l_3 and from l_3 to l_4. The timed automaton may wait until $c_1 \leq 90$ in location l_3 and leave the earliest at $c_1 \geq 80$. Only then, the outgoing message is sent.

In order to maintain a separation of concerns [MSKC04], i.e., not to mix the component's functional behavior with its self-healing behavior, the behavior of each component is specified by at least two timed automata. One timed automaton specifies the functional behavior. The other specifies the self-healing behavior, i.e., error detection and the resulting initiation of reconfiguration.

2.2.3 Timed Component Story Diagrams

In MechatronicUML, we use structural reconfiguration to adapt the system behavior. This means, the system behavior is adapted at runtime by deleting or creating software components. The system's architecture is specified by an initial deployment diagram and a set of reconfiguration rules [EHH+13]. A reconfiguration rule transforms one deployment diagram into another deployment diagram.

In this thesis, we model reconfiguration by the change of the system architecture based on the MechatronicUML component model (cf. Section 2.2.1). Therefore, MechatronicUML employs *component story diagrams* as introduced in [THHO08]. Component story diagrams specify architectural transformations on deployment diagrams. They are based on UML activities. Each activity specifies a transformation of the deployment diagram in the form of a *component story pattern*. Component story patterns are an extension of story patterns that allow for using the concrete syntax of deployment diagrams for specifying reconfigurations.

We extend component story diagrams to *timed component story diagrams* (TCSD) by adding time intervals, namely *durations*, to the component story patterns. The time interval specifies the minimum and maximum time units it takes to apply the timed component story pattern (TCSP) to a deployment diagram during runtime. This duration includes the time for created components to start up and for deleted components to shut down.

The duration of the TCSD is computed from the durations of its TCSPs. For this, the paths with the shortest and longest duration must be determined. In order to get a longest duration that is not infinite, the maximum duration of a TCSD must be bounded by a finite value.

Each TCSD has its own internal clock that starts with zero, when the execution of the TCSD is started. The duration of the TCSD refers to this internal clock.

Figure 2.12 shows a TCSD that creates an embedded component instance co:Coordination if it is not yet present, and connects it to the ports provider and coordinator of the current embedding component instance.

The time constraint $[90, 105]$ in Figure 2.12 specifies that at least 90 and at most 105 time units elapse from the time when the application of the rule is initiated and the last change in the deployment diagram that completes the rule. We assume that the system developer knows this information.

For the specification of reconfigurations in TCSPs, we consider deployment diagrams as graphs, i.e., components as nodes and connectors as edges. We then model reconfiguration as a graph transformation rule. A graph transformation rule consists of a left hand side and a right hand side. The left hand side identifies the part of the deployment diagram in which the rule can be applied, i.e., the graph contains an isomorphic image of the left hand side as a subgraph. The right hand side defines the result of the application, i.e., the changes to be

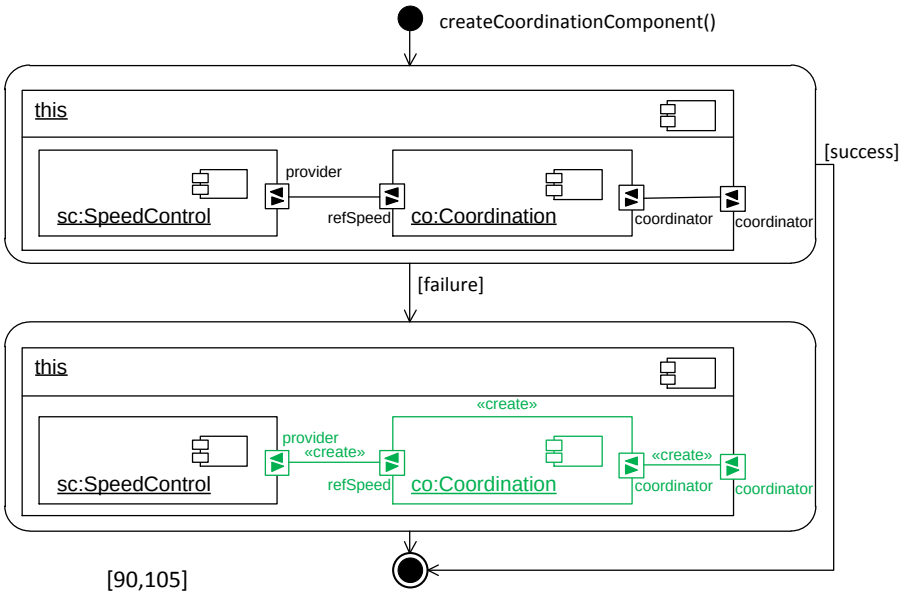


Figure 2.12: TCSD creating a coordination component

made to the subgraph such that it is an isomorphic image of the right hand side. For a short hand notation, left and right hand side are depicted in one single graph. Instances that are created during rule application are drawn in green and are annotated with the stereotype «`create`». Instances that are destroyed are drawn in red and are annotated with the stereotype «`destroy`».

A TCSP is executed by first matching the left hand side to the deployment diagram. Then all parts that are part of the left hand side but not of the right hand side are deleted. Thereafter, all parts that are part of the right hand side but not of the left hand side are created.

Then, the right hand side of the TCSP is instantiated and links between the instances in the match of the left hand side and the rest of the deployment diagram are redirected to the corresponding ports of the right hand side. Afterwards, the now disconnected instances matched by the left hand side are deleted.

We assume that no concurrent reconfigurations occur, i.e., all aspects of the reconfiguration are described as single TCSDs or sequences of TCSDs. Thereby, the possibility of unintentional interference between TCSDs is excluded. Otherwise, the creation of arbitrary intermediate architecture or an over- or underestimation of the execution time would be possible.

2.2.4 Time

All operations in a mechatronic system have to meet real-time requirements. Therefore, in MechatronicUML, stateful behavior and reconfiguration is constrained by time constraints. These time constraints specify the start time and duration of operations, and communication delays.

The start time of operations depends on the duration and start time of preceding operations and communication delays. The duration of operations is determined by the instructions in the program code, the hardware, the system is deployed on and by the software that abstracts the hardware from the real-time software.

The computing hardware is the basis for the durations of operations in a system. The hardware introduces many different parts that all affect the execution of instructions. It is obvious that the processor speed and the processor cache size have a direct impact on the speed of executions. However, this speed is for example also influenced by the bus system and the memory speed. Both affect memory latency [BO10].

Several layers may exist between the hardware and the real-time software several [TG98]. Each layer may introduce delays and affect the duration of operations. A typical computer architecture consists for example of the following layers: hardware, firmware, assembler, kernel, operating system [TG98]. The OCM (cf. Section 2.1.4) also defines a layered architecture of hardware and software components that each affect the time properties of the system.

Connections between components may have varying delays [BD09]. The delay is the time required for a message to propagate from source to target. The delay depends on a variety of parameters, e.g., queuing time, processing delay, propagation time, and transmission time. The propagation time depends on the physical properties of the connection. The speed usually has a value of approx. $10^{-8}\frac{m}{s}$ [BD09]. Propagation times are also affected by changing environment conditions like changing temperatures. A further uncertainty is introduced by using buses for data transmission. Depending on the load of the bus transmission times may vary. Consequently, communication delays have to be expressed by an interval that specifies minimum and maximum delays. This also applies to the start time and duration of operations in components, because these operations may depend on data that is transmitted via a connection to another component.

Several kinds of analyses may be applied to include the above mentioned times and durations into system models. The durations of reconfigurations, for example, are determined by a worst case execution time analysis [TGS06, THMvD08]. The worst-case execution time is the longest time needed to finish an operation. The reconfigurations are integrated into real-time statecharts as side effects where they affect the time constraints of the real-time statecharts. The time constraints in real-time statecharts are for example further influenced by communication with other system components.

2.3 Safety

Mechatronic systems are often safety-critical systems. This means, they are applied in environments where property may be damaged or people may be harmed as for example in airplanes, trains or medical devices. *Dependability* provides a means to characterize safety-critical systems with respect to safety and security-related aspects.

Figure 2.13 illustrates the complete taxonomy of dependable computing as defined by Avizienis et al. [ALRL04]. It comprises attributes, threats and means of dependability. We explain these aspects in more detail below.

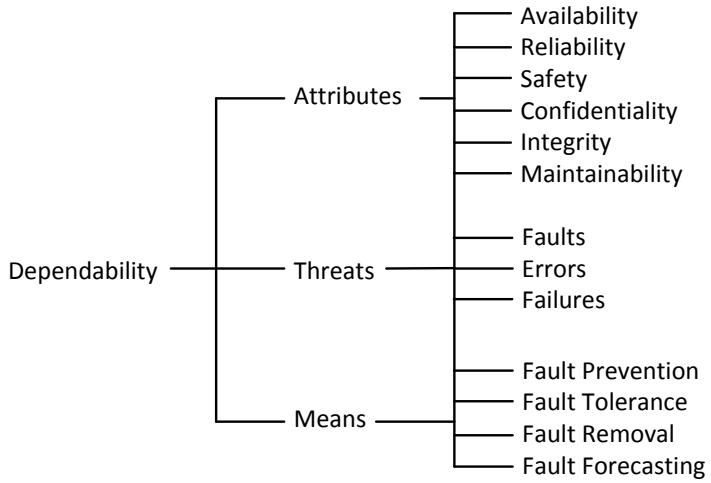


Figure 2.13: Taxonomy of dependable computing [ALRL04]

The concept of dependability is a general term covering the following six attributes:

- Availability: to be ready to deliver correct service
- Reliability: to continue a correct service
- Safety: the non-occurrence of critical failures
- Confidentiality: the absence of unauthorized disclosure of information
- Integrity: the non-occurrence of invalid system modifications
- Maintainability: the ability to be modified and repaired

This work particularly focuses on the safety of self-healing mechatronic systems and on how to maintain it.

The attributes confidentiality and integrity exclusively address the concept of security. *Security* and *safety* are closely related. They both deal with threats – safety "with threats to life and property", security "with threats to privacy" [Lev95]. In this work, we particularly focus on safety and neglect the concept of security.

Threats to safety particularly focus on the propagation of defects through the system. *Failures* are the externally visible deviation from the component's behavior. *Errors* are the manifestation of a *fault* in the system state, whereas a fault is the cause of an error.

The *means* to attain dependability are techniques that aim at maintaining or improving the dependability of a system. They are grouped into four categories [ALRL04]:

- Fault prevention: faults do not occur
- Fault tolerance: avoid failures in the presence of faults
- Fault removal: reduce the number of faults
- Fault forecasting: estimate future faults

All these techniques may be applied at design time. Faults may for example be prevented and removed from the system by using reliable hardware components. Faults may be forecasted by hazard analysis, which considers the probabilities of faults. Fault tolerance techniques enable the system to deliver its service even in the presence of faults [Sto96]. A special case of fault tolerance is self-healing.

The system's safety may be threatened if the system does not deliver its specified service. For example, if a RailCab drives at another speed than intended, this could lead to collision or derailment.

Such situations like the collision are called *accident*: "An *accident* is an unintended event or sequence of events that causes death, injury, environmental or material damage." [Sto96].

A situation that may, under the right circumstances, lead to an accident is a hazard. "A *hazard* is a situation in which there is actual or potential danger to people or the environment" [Sto96]. A hazard thus specifies a dangerous situation as, e.g., a RailCab, which is driving with a wrong speed.

Hazards cannot always be prevented, even though the system has been implemented correctly. They must therefore be taken into account during system design. The occurrence probabilities and causes of hazards are computed by hazard analysis (cf. Section 2.4). Hazard analysis is used to implement and construct a system such that hazards only occur with a specific probability [Lev95].

However, the occurrence probability of a hazard alone does not provide sufficient information to assess the criticality of a hazard. In order to assess the criticality of a hazard, it is related to its occurrence probability *and* consequences. It would be more dangerous if, for example, two RailCabs collided that are filled with passengers compared to the case if two empty RailCabs collided. In the first case, the severity of the hazard implies multiple deaths compared to no deaths in the second case. Thus, the wrong distance between two RailCabs is particularly critical if the RailCabs have loaded passengers.

A means to measure the criticality of a hazard is the *risk*. "Risk is a combination of the frequency or probability of a specified hazardous event and its consequence" [Sto96].

Risk provides a means to classify the acceptability of hazards during system design. The developer specifies a maximum risk for the system under development. All hazards whose risk does not exceed the maximum are acceptable. If there are hazards that are not acceptable with respect to the system requirements, the system must be modified such that all hazards become acceptable. This may be achieved by several techniques as for example hazard reduction [Lev95]. Reducing a hazard means decreasing its occurrence probability.

Hazards are caused by faults that occur in system components. Faults are classified by different viewpoints [ALRL04]. They may, for example, be classified by their dimension (software or hardware), objective (malicious or non-malicious), or persistence (permanent or transient). In the domain of real-time systems, persistence is of particular interest. A transient fault that only exists for a specific time period may have a less negative impact on the system or even be completely tolerated. In contrast, a permanent fault will always have negative impact if it is not corrected.

We use the terminology of Avizienis et al. [ALRL04] to specify the propagation of faults through the system. An externally visible deviation of the component behavior from its intended function is called a *failure*. Such failures happen as a result of a chain of events (cf. Figure 2.14) in a system component: First, a *fault* happens, e.g., a sensor part is broken and the sensor delivers values that deviate from the real values. This fault then causes an *error* - a deviation from a correct system state, e.g., a wrong value in the system memory. This error causes a failure, e.g., due to the wrong value in the memory the system outputs a wrong value for the control of an engine. This failure may lead to another fault in a component or system that is connected to this component or system.

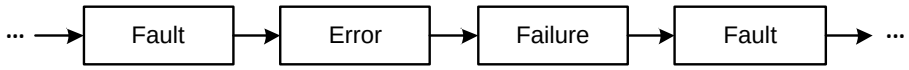


Figure 2.14: Fault error failure chain

An important class of faults are byzantine faults. A byzantine fault occurs if multiple processors of a component in a distributed system deliver wrong results but the results conform to the component specification. In the field of self-healing, it depends on the fault detection mechanism whether these faults are detected correctly. For byzantine faults there exists an approach for fault tolerance, namely byzantine fault tolerance [CGR11].

Another class of failures, stop failures, lead to a system crash. In this case, only a fallback arrangement or a restart will help.

2.4 Hazard and Risk Analysis

Hazard analysis is one step to ensure the safety of a system and control safety threats. Hazards are identified, assessed, and if necessary removed or reduced in order to guarantee that hazards only threaten the safety of a system by an acceptable degree.

Hazard analysis is divided into three general tasks [Lev95]:

(1) Development: Potential hazards are identified and assessed at design time and, if necessary, actions are taken to control or eliminate the hazards.

(2) Operational Management: An existing system is analyzed to identify and assess hazards with the goal to, e.g., improve the safety of the system.

(3) Certification: The safety of an existing or planned system is demonstrated to get it accepted by authorities or the public.

In this thesis, we focus on the task *development*. This means, we aim at analyzing hazards of a non-existing system, which is under construction. The goal of this task is to make the developed system safer. The development task is divided into subtasks [Lev95]:

(1) Identify hazards.

(2) Show the absence of specific hazards.

(3) Determine damages that result from hazards.

(4) Evaluate the causes of the hazards.

(5) Identify procedures to eliminate, minimize or control the hazards.

(6) Find concrete actions to eliminate hazards.

(7) Find concrete actions to control hazards that cannot be eliminated.

(8) Evaluate hazard control.

(9) Provide information for quality assurance.

(10) Evaluate the modification which are to be applied on the system.

In this thesis, we address Step (8) – the evaluation of hazard control. Self-healing is a method for hazard control as it aims at reducing the occurrence probabilities of hazards. To evaluate self-healing operations, we apply existing methods to compute the causes and occurrence probabilities of hazards. A common method for this evaluation is fault tree analysis [Lev95].

2.4.1 Fault Tree Analysis

Fault tree analysis [VGRH81, Sto96] is a graphical method for qualitative and quantitative assessment of hazards. It starts with an event that represents a hazard. The fault tree works backwards from this hazard to the causes of the hazard. Events that are related to the top-event are connected by Boolean operators like AND or OR. This is repeated until the basic events of the hazard are found.

Figure 2.15 shows an example of a simplified fault tree. The top event "wrong position on track" represents the hazard of the RailCab being located on another position on the track than indented. The direct causes for this hazard are a wrong speed of the RailCab or a wrong steering angle. A wrong speed is caused if both speed sensors fail at the same time.

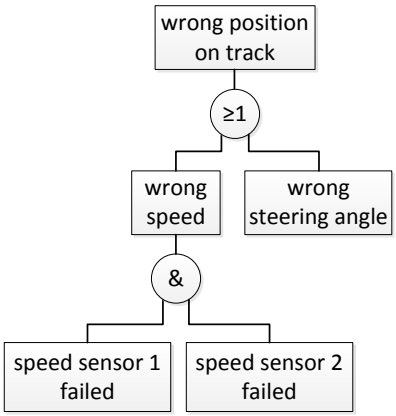


Figure 2.15: Fault tree

Fault tree analysis provides qualitative and quantitative results. *Qualitative fault tree analysis* determines the combinations of basic events that lead to a hazard, namely *cut sets*. If they occur together, the hazard will occur. *Minimal cut sets* (MCS) are a special case of cut sets. A hazard will only occur if all basic events of a MCS occur. If one basic event does not occur, the hazard will not occur. *Quantitative fault tree analysis* determines the occurrence rate or probability of a hazard. This is based on the occurrence rates or probabilities of the basic events of the MCS. In this thesis, we use occurrence probabilities. Occurrence probabilities are computed from occurrence rates.

The occurrence rates of hardware components are determined by testing or by logging failures during the lifetimes of systems [Lev95]. In this thesis, we assume that these rates are already given for example by data sheets. Data sheets are one established method to acquire occurrence rates of basic events. These sheets are provided by manufacturers of hardware components as for example an optical transceiver [Tec08] or hard disk drives [LLC08]. These data sheets provide several means for reliability. Among others, these are the *mean time to failure* (MTTF) and the failure rate per 1000 hours. The MTTF specifies the

estimated time of operation until the first failure occurs [Sto96]. The failure rates or probabilities, which are used for fault tree analysis are derived from the MTTF.

The following formula computes the MTTF from the failure rate λ and a period of time t [Sto96]:

$$MTTF = \int_0^\infty e^{-\lambda t} dt = \frac{1}{\lambda}$$

The failure rate can thus the computed by the multiplicative inverse of the MTTF.

The occurrence probability of a failure of a component is computed from the reliability of this component [Sto96]. A failure is the complementary event of the reliable system. Thus the occurrence probability $F(t)$ that a system may not work correctly over a given period of time t is the complement of the reliability over this period t [Sto96]:

$$F(t) = 1 - R(t)$$

The reliability over a period of time t is computed from the failure rate λ by [Sto96]:

$$R(t) = e^{-\lambda t}$$

2.4.2 Fault Tree Analysis in Self-optimizing Mechatronic Systems

Giese et al. [GTS04, GT06] developed a fault tree analysis for self-optimizing mechatronic systems. This analysis takes the special characteristics of self-optimizing mechatronic systems like cyclic structures and reconfiguration into account, which cannot be analyzed by standard fault tree analysis. To complement the development of software for self-optimizing mechatronic systems, this fault tree analysis has been integrated into MechatronicUML.

The approach of Tichy et al. [GT06] uses UML deployment diagrams extended by hardware ports to specify and analyze the influence of hardware faults to the software components. Component types and hardware nodes are enhanced by Boolean formulas that specify the failure propagation inside the component. The failure propagation between hardware nodes and component instances and between component instances is derived automatically from the connectors of the deployment diagram. Hazards are specified by fault trees. The leaves of the fault trees correspond to outgoing failures of the components of the system structure. Qualitative and quantitative fault tree analysis is performed based on Binary Decision Diagrams. The minimal cut-sets (or prime implicants) can be computed using different approaches (exact ones [vOQ55, McC56, RD97] or heuristics [BSMH84, Rud84, Sed08]).

The usage of Boolean formulas instead of fault trees allows for representing and analyzing cycles and common cause failures. This is particularly important for analyzing mechatronic systems, because each mechatronic system has at least one cycle: the control loop (cf. Section 2.1.1).

Errors and failures are typed using an extensible failure classification like the one from Fenelon et al. [FMNP94]. This allows for the abstraction from concrete errors and failures. This in turn allows for a more precise specification of failure propagation between components. We distinguish the general error and failure classes *value*, *service* and *timing*. A value error specifies that a value deviates from a correct value, e.g., an erroneous value in the memory of a component. A service error specifies that no value at all is present, e.g., a component crashed. A timing error occurs if a value is provided too early or too late.

Error and failure types are ordered by a hierarchy. Upper elements are a generalization of lower elements and vice versa. Figure 2.16 shows an example of a failure type hierarchy with the failure types mentioned above. The failure class *value failure*, for example, is a generalization of the failure classes *coarse value failure* and *subtle value failure*.

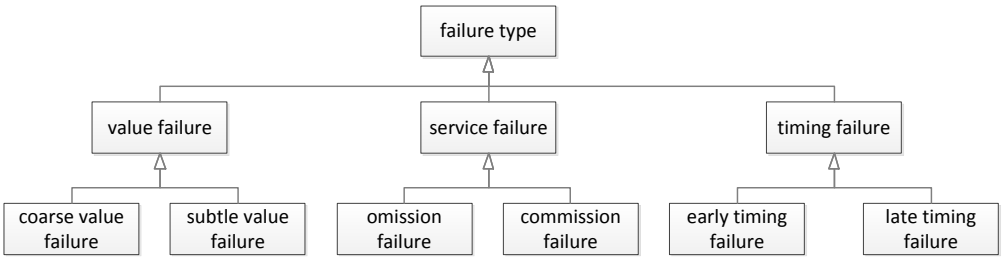


Figure 2.16: Failure type hierarchy

Errors and failures are represented by error and failure variables. Error and failure variables are named according to the following scheme: $e_{k,ft}$ and $f^{d}_{k.p,ft}$ for component type k, port type p, error or failure class ft, and $d \in \{\mathrm{i}, \mathrm{o}\}$. i and o specify the direction of failures – i stands for incoming and o for outgoing. In a deployment diagram, error and failure variables are instantiated when instantiating component types. The notation, described above, holds for component instances and port instances analogously. This instantiation makes all error and failure variable instances unique.

Figure 2.17 shows a deployment diagram with a visualization of the Boolean formulas that specify the failure propagation. Error variables are drawn as circles and failure variables as rectangles. Operators are depicted by circles. Arrows indicate the orientation of the propagation.

In the example of Figure 2.17, the value error $e_{gps,v}$ in the component instance gps:GPS causes the outgoing value failure $f^{o}_{gps.p1,v}$ at port p1 of gps:GPS. The outgoing value failure $f^{o}_{dg.p3,v}$ at port p3 of component instance dg:DistGPS occurs if at least one of the incoming value failures $f^{i}_{dg.p1,v}$ or $f^{i}_{dg.p2,v}$ occurs.

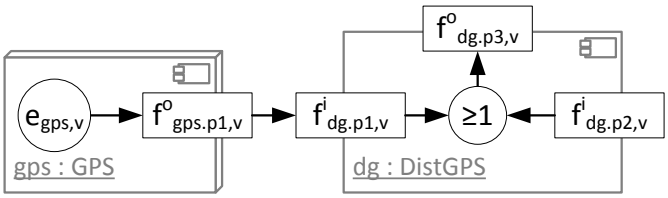


Figure 2.17: Deployment diagram with failure propagation model

Hazards are modeled by Boolean formulas that specify the combinations of outgoing failures of a deployment diagram that cause hazards. The Boolean formula that defines the hazard is represented by a fault tree. This fault tree connects the outgoing failures of the deployment diagram to the hazard, which is the top event of the fault tree.

Figure 2.18 shows the fault tree for the hazard wrong distance that represents the hazard of a wrong distance between two RailCabs in a convoy. The causes of this hazard are an outgoing value failure of port p4 of the distance control component dc or an outgoing value failure of port p3 of the steering angle control component sa. These failures are represented by the failure variables $f^o_{dc.p4,v}$ and $f^o_{sa.p3,v}$.

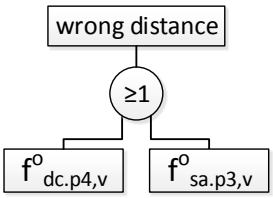


Figure 2.18: Fault tree specifying the hazard wrong distance

Reconfiguration is addressed by analyzing all possible architectures of the system. Therefore, the reachable architectures of the system are computed with the help of real-time statecharts and graph transformation rules [GT06]. The failure propagation of a component is extended by the information about the architectures and the according failure propagation. The architectures are stored in Boolean variables that determine the failure propagation according to the respective architecture. The results of the reachability analysis are mapped to constraints over the Boolean variables of the architectures. The resulting Boolean formula represents a symbolic encoding of all possible failure propagations that result in the hazard. This model is used to compute which cut sets cause a hazard on which architecture. Further, the best and worst architecture with respect to hazard occurrence probability is determined [GT06].

This hazard analysis analyzes all possible system architectures. However, it does not take into account where failures are located at the point in time when a reconfiguration is executed. It is therefore not applicable for analyzing self-healing operations.

2.4.3 Risk Analysis

Risk analysis combines hazard frequencies and consequences to compute the *risk* of a hazard [Sto96] as introduced in Section 2.3. Thereby, risk allows to classify a hazard by its criticality.

There exist many standards for the quantitative classification of risk. They all have the combination of the occurrence probability of a hazard and the severity of its consequences in common. The "International Standard IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems" [Com98] focuses on the safety of computer controlled systems. This standard classifies hazard occurrence probabilities and hazard consequences into categories. The combinations of the categories yield the risk classes.

The result of risk analysis is the risk class of a hazard. For risk classes categories are used. The standard IEC 61508 proposes the risk classes I, II, III, and IV ranging from the most serious accident (I) to the least serious accident (IV). Table 2.1 shows an example of a possible relationship between risk classes and severities and frequencies of a hazard. The standard does not specify concrete values for the combination of severity and frequency.

Frequency	**Categories**			
	Catastrophic	Critical	Marginal	Negligible
Frequent	I	I	I	II
Probable	I	I	II	II
Occasional	I	II	III	III
Remote	II	III	III	IV
Improbable	III	III	IV	IV
Incredible	IV	IV	IV	IV

Table 2.1: Risk classes from IEC 61508 [Com98]

The concrete values for the severity and frequency categories are not given by the standard IEC 61508 [Com98]. The standard only provides a guideline for the development of safety-critical systems. Consequently, results of analyses of different analyzers may differ even though the system has similar safety-related characteristics.

Hazards mostly cannot be removed completely from a system. The goal is to make hazards acceptable. The acceptability of a hazard is a trade-off between the benefits of the risk level and the costs needed to reduce this risk level. This principle is called $ALARP$[4][Sto96]. The IEC 61508 [Com98] categorizes risk levels into three categories as illustrated in Figure 2.19.

The unacceptable region of the diagram represents hazards with a risk that is so great that it is unacceptable. Hazards whose risk is located in the broadly acceptable region can be neglected because their risk is very low. Between these

[4] As low as is reasonably practicable

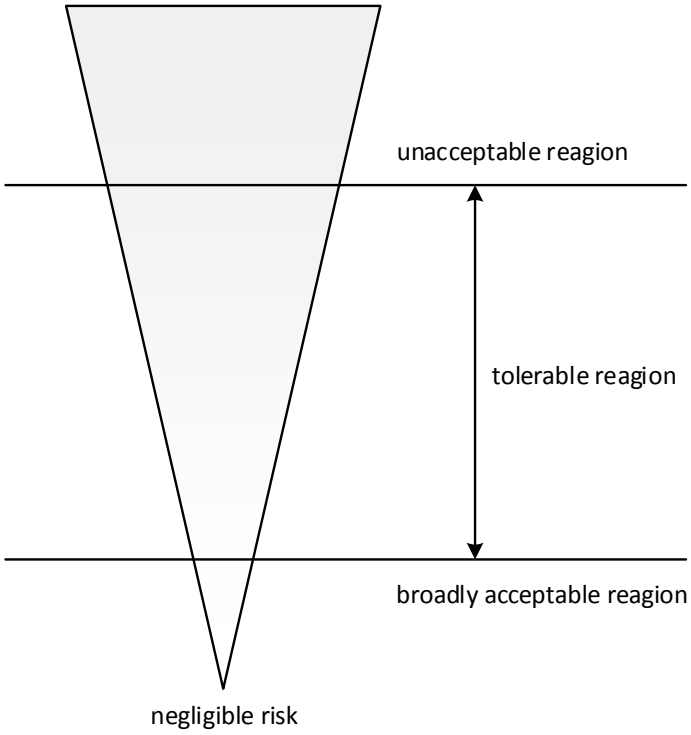


Figure 2.19: Levels of risk [Sto96]

two regions lies the tolerable region. Risks that lie within this region may be acceptable under certain circumstances. A risk within this region is tolerable if it is as low as reasonably possible. This means, if the risk can be educed easily, it is not acceptable in this region. If, however, the reduction has high costs, it will be accepted.

Different systems have different safety requirements. Obviously, the safety requirements for a toy car are far lower than the safety requirement for a real car. These varying safety requirements are reflected by the concept of safety integrity:

"Safety integrity is the likelihood of a safety-related system satisfactorily performing the required safety functions under all the stated conditions within a stated period of time"[Sto96].

There exist many standards for the quantification of safety integrity [Sto96]. They have in common that safety integrity levels (SIL) are allocated to a system. The classification of SIL varies greatly among the different standards. The international community converged on a classification of four levels as shown in Table 2.2. These SIL are specified in the international standard IEC 1508. They specify the maximum number of failures for a given period of time.

SIL are specified for two classes of systems. Systems that operate in *continuous mode* and systems that operate in *demand mode*. For systems in continuous

Safety integrity level	Continuous mode of operation (probability of failure per year)	Demand mode of operation (probability of failure on demand)
4	$\leq 10^{-5}$ to $< 10^{-4}$	$\leq 10^{-5}$ to $< 10^{-4}$
3	$\leq 10^{-4}$ to $< 10^{-3}$	$\leq 10^{-4}$ to $< 10^{-3}$
2	$\leq 10^{-3}$ to $< 10^{-2}$	$\leq 10^{-3}$ to $< 10^{-2}$
1	$\leq 10^{-2}$ to $< 10^{-1}$	$\leq 10^{-2}$ to $< 10^{-1}$

Table 2.2: Target failure rates for the safety integrity levels of draft IEC 1508 [Sto96]

mode, SIL are quantified by failures per year. For systems that operate on demand, SIL are quantified by failures per demand [Sto96].

The SIL, which is allocated to a system, is based on its risk classification. For subsystems, individual SIL may be assigned.

In summary, the main part of risk analysis is the computation of hazard occurrence probabilities and MCS. The risk is computed by the product of the hazard occurrence probability and severity. The severity of hazards mostly cannot be modified. However, the system design can be adjusted to meet certain thresholds of hazard occurrence probabilities. In this work, we therefore focus on hazard occurrence probabilities when referring to safety requirements.

2.5 Summary

In this chapter, we introduced the foundations, which are necessary to understand the methods which have been developed in the course of this thesis. First, we gave an introduction of self-healing mechatronic systems in Section 2.1. In Section 2.2, we presented MechatronicUML, a graphical modeling language which comprises diagrams for modeling the architecture, reconfiguration, and real-time behavior of the software of mechatronic systems. In this thesis, we will use MechatronicUML for modeling software of mechatronic systems. In Section 2.3, we presented means to assess the safety and in particular potential threats to the safety of a system. One of these threats are hazards, which cannot be removed completely from technical systems. In Section 2.4, we presented methods which are used to compute the causes and occurrence probabilities of hazards. With these methods, the developer has a means to guarantee that hazards only occur with acceptable probabilities. In particular, we presented the hazard analysis of Giese et al. [GTS04, GT06] which has been integrated into MechatronicUML. It allows for analyzing hazards in mechatronic systems. However, it does not take into account where failures are located at the point in time when a reconfiguration is executed. It is therefore not applicable for analyzing self-healing operations. However, we will use this analysis to compute

MCS and hazard occurrence probabilities as part of the analysis of self-healing operations. In the next chapter, we show how we model the propagation of failures including failure propagation times such that we are able to compute the location of failures in the systems at specific points of time.

3 Modeling Timed Failure Propagation

In this chapter, we introduce the models that are the basis of our analysis of self-healing operations which will be presented in Chapter 5. The analysis uses failure propagation models that are extended by propagation times. We call these failure propagation models *timed failure propagation graphs* (TFPG).

The propagation of failures through the system during specific time intervals is computed by a formal analysis. To apply this formal analysis to TFPGs, we formalize TFPGs the syntax and semantics of TFPGs in Section 3.4. Our definition of TFPGs is based on the component structure of the system, because AShOp is a component-based analysis. We therefore give a formal definition of the component architecture of the system in Section 3.2. We further present a formal definition of the system behavior in Section 3.3, because the propagation times of TFPGs are computed from the reachable behavior of the system. We start with the description of an application example, which we will use to illustrate the models and analyses, which we present in this thesis.

3.1 Example

We explain the approaches and models of this thesis using the speed control subsystem of the RailCab as an example. The speed control subsystem is a safety-critical part of the RailCab, particularly when driving in a convoy. A failure in this subsystem may lead to a wrong speed, which is a hazard. A wrong speed may result in a collision or derailment and thus cause severe injuries or property damage. Figure 3.1 shows the deployment diagram of the speed control subsystem.

The speed of a RailCab is controlled by the speed controller sc:SpeedCtrl. It computes the electric current, which is applied to the linear drive. The electric current depends on several parameters: the position of the RailCab on the track, the distance to other RailCabs in the convoy, and the reference speed and distance specified by the convoy leader. The position of the RailCab is computed by pos:PosCalc from the signals of the two speed sensors s1:SpeedSensor and s2:SpeedSensor and a GPS-signal provided by the GPS-sensor gps:GPS. The distance to another RailCab is measured by the distance sensor dr:DSensor and computed from GPS-data by dg:DistGPS. The latter component computes the distance from the position data of gps:GPS and the GPS data of the adjacent RailCabs which is received via wireless network by the components wlan:WLAN and ref:ReferenceData. ref:ReferenceData further provides reference data, e.g., the

convoy speed from the convoy leader. strt:Strategy determines the drive modes of the RailCab. In our example, it sets fast and slow drive modes. Therefore, strt:Strategy forwards the drive mode to different component instances of the speed control subsystem. The example of Figure 3.1, only shows the connection between strt:Strategy and pos:PosCalc. We will have a closer look at pos:PosCalc in Chapter 4.

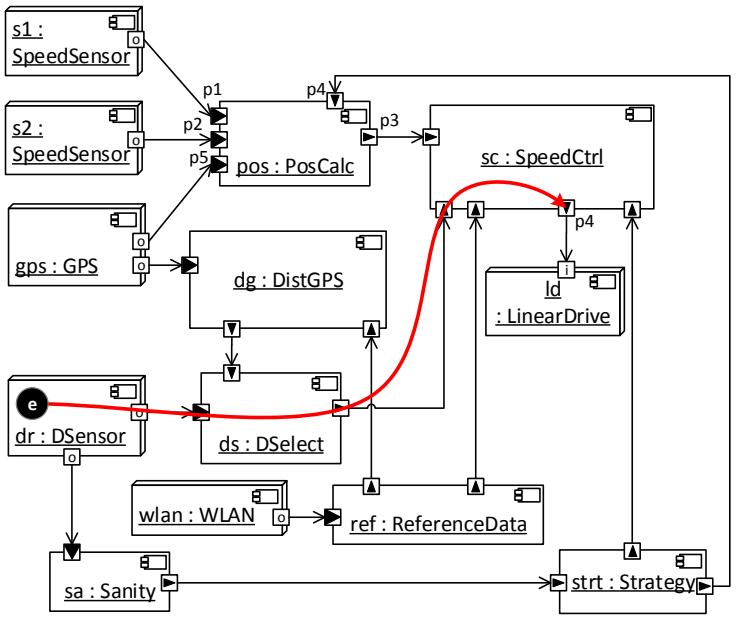


Figure 3.1: Deployment diagram of the distance control subsystem

The labels of the ports of the component instances pos:PosCalc and sc:SpeedCtrl show the names of the labeled ports. They will be referred to later in this work.

An error in the distance sensor dr:DistSensor will lead to an outgoing failure of the speed controller sc:SpeedCtrl which causes the hazard wrong speed. Figure 3.1 illustrates this by a red arrow.

We specify hazards by Boolean formulas over outgoing failures as introduced in Section 2.4.2. The hazard wrong speed occurs, if a wrong value is output by sc:SpeedCtrl. This hazard is thus defined by the Boolean formula

$$wrong\_speed \Leftrightarrow f^{o}_{sc.p_4,v}.$$

This means, we associate the hazard wrong speed to the outgoing value failure at port p_4 of the speed controller sc:SpeedCtrl.

To prevent the outgoing failure $f^{o}_{sc.p_4,v}$ of the speed controller in case of an error in the distance sensor, the component instance sa:Sanity monitors the sanity of the signal from the distance sensor dr:DSensor. If sa:Sanity detects a failure in this signal, it forwards a reconfiguration proposal to strt:Strategy. Next, strt:Strategy triggers a self-healing operation that is specified by the TCSD depicted in Figure 3.2.

The self-healing operation removes the connectors between the component instances of the types DistGPS and DSelect, and SpeedCtrl and DSelect. It further creates a new connector between component instances of the types DistGPS and SpeedCtrl. Thereby, the self-healing operation disconnects dr:DSensor and ds:DSelect from the system. After the application of the self-healing operation, the distance between two vehicles will be computed from the values of gps:GPS and the reference data from the second vehicle only. The execution of the TCSD takes between 79 and 86 time units.

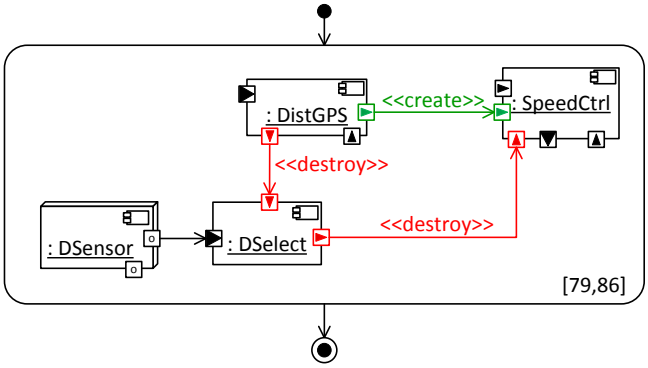


Figure 3.2: TCSD specifying a self-healing operation

3.2 System Architecture

The TFPG definition is based on component-based architectures. We therefore give a formal definition of a component-based architecture and reconfiguration rules. TFPGs may be applied to all component models that use components which communicate via ports. In this work, we use the MechatronicUML component model as introduced in Section 2.2.1. The definition of reconfiguration rules is required to analyze the effect of self-healing operations on the propagation of failures in AShOp (cf. Chapter 5).

We first define the component specification of the system which gathers all component types and hardware nodes, which may occur in the system. We define a component specification as a type graph as defined in [EEPT06].

Definition 3.2.1 (Component Specification)
We define the component specification sys *as a type graph* $sys = (V_{sys}, E_{sys}, s_{sys}, t_{sys})$ *over a set of* component types K*, a set of* hardware nodes H*, a set of* port types P_K*, a set of* hardware ports P_H*, and a set of* connector types L *with*

- $V_{sys} = K \cup H \cup P_K \cup P_H$,
- $E_{sys} = L$
- $s_{sys} : E_{sys} \to V_{sys}$ *the source function, and*

- $t_{sys} : E_{sys} \to V_{sys}$ *the target function.*

A component specification specifies a system by defining which component, port and connector types and hardware nodes exist, which port types are associated to which component types and hardware nodes, and which connector can connect which port types.

We define a deployment diagram as a typed graph as defined in [EEPT06] based on the type graph of the component specification.

Definition 3.2.2 (Deployment Diagram)
Let $sys = (V_{sys}, E_{sys}, s_{sys}, t_{sys})$ *be a component specification with* $V_{sys} = K \cup H \cup P_K \cup P_H \cup L$. *We define* $\overline{K}$ *the set of* component instances *of* K, $\overline{P}_K$ *the set of* port instances *of* P_K, $\overline{H}$ *the set of* hardware instances *of* H, $\overline{P}_H$ *the set of* hardware port instances *of* P_H, *and* $\overline{L} \subseteq (\overline{P}_K \times \overline{P}_K) \cup (\overline{P}_H \times \overline{P}_K)$ *the set of* connector instances *of* L.

A deployment diagram $w = (G_{conf}, type)$ *of a component specification* sys *is a typed graph, where* $G_{conf} = (V_{conf}, E_{conf}, s_{conf}, t_{conf})$ *is a graph with the vertices* $V_{conf} = \overline{K} \cup \overline{H} \cup \overline{P}_K \cup \overline{P}_H$, *the edges* $E_{conf} = \overline{L}$, *and the source and target function* s_{conf} *and* t_{conf} *and type is a graph morphism typing* G_{conf} *over* sys.

A deployment diagram is defined by a set of instances of the component and hardware types of a component specification, and connections between port instances which are realized by connector instances.

In MechatronicUML, reconfiguration rules are modeled by timed component story diagrams (TCSD) (cf. Section 2.2.3). In this thesis, we consider TCSDs that consist of one TCSP only.

Definition 3.2.3 (Timed Component Story Pattern)
Let sys *be a component specification.*

A timed component story pattern *is a tuple* $r = (LHS, RHS, d)$. LHS *(left hand side) and* RHS *(right hand side) are typed graphs as defined in [EEPT06] typed over* sys. $d = [d_{min}, d_{max}]$ *with* $d_{min}, d_{max} \in \mathbb{R}_{\geq 0}$, $d_{min} \leq d_{max}$.

TCSPs are defined for component types and hardware nodes. This means, TCSPs are specific to the component specification of a system but not specific to a deployment diagram. TCSP application, of course, takes place on the deployment diagram rather than the component specification.

d specifies the minimum and maximum duration of time needed to transform LHS into RHS.

The application of a TCSP to a deployment diagram requires identifying a matching of the TCSP's left hand side to the deployment diagram.

Definition 3.2.4 (Matching)
Let w be a deployment diagram and $r = (LHS, RHS, d)$ a TCSP. Let w and r be typed over a component specification sys. We define a matching $m(w, r)$ *of the TCSP r on the deployment w as a subgraph isomorphism [EEPT06] of LHS on w.*

An LHS may be matched to several subgraphs of a configuration. In the case of self-healing operations, we allow only one matching, because the self-healing operation needs to be applied such that it affects the failure propagation as intended. We therefore assume that the developer specifies a self-healing operation that only has one matching in the configuration to which it is applied.

The number of matchings of an LHS in a configuration may be computed by isomorphism checks. Unfortunately, isomorphism checks are NP-complete, because all possible mappings of all nodes in the LHS and the existence of all required edges must be checked. However, in timed component story patterns (cf. Section 2.2.3), and story patterns in general, this complexity can be reduced significantly in most cases [SWZ95]. Story patterns are typed graphs. They usually contain many different types of nodes and edges. Only nodes and edges of the same types may be matched. Consequently, the graph search terminates quickly in case it started in the wrong node. Further, the developer may specify node attributes, e.g., an object name, and thereby define parts of the matching in the configuration itself. This provides a starting point for the subgraph search. Additionally, the nodes and edges are accessed by hashing. This means, request are processed efficiently even in large graphs [SWZ95].

3.3 System Behavior

We specify the real-time behavior of component types and component instances by real-time statecharts as we have explained in Section 2.2.2. The formal semantics of real-time statecharts is defined by mapping real-time statecharts to timed automata [BY03]. In the remainder of this thesis, we will use timed automata to model real-time behavior, because the algorithms which were used and developed in the course of this thesis are applied on time automata. Timed automata contain less syntactic elements than real-time statecharts. For example in contrast to real-time statecharts, timed automata do not contain deadlines. By using timed automata for modeling our running example, the results of the used algorithms become more comprehensible.

A timed automaton is a finite state machine that is extended by a set of real-valued variables called clocks and constraints over these clocks. Thus, timed automata allow for defining time-dependent behavior. As in real-time statecharts, the specified behavior does not only depend on inputs but also on the

point in time when these inputs are received. Real-time statecharts are transformed into timed automata as presented in [BGHS04].

Below, we formalize timed automata based on the definition of Bengtsson and Wang [BY03]. We extend this definition by variables and operations and guards on these variables, which were introduced informally by Bengtsson et al. [BGK+96]. We use the variables for passing parameter values between timed automata, because the definition of timed automata of [BY03] does not support messages with parameters. We further extend timed automata by side effects that represent the execution of TCSPs.

Definition 3.3.1 (Timed Automaton)
Let $\mathcal{R}$ be a set of TCSPs. A timed automaton A is a tuple $A = (L, l_0, C, V, \Sigma, R, E, I)$ over $\mathcal{R}$ where

- *L is a finite set of locations,*
- *$l_0 \in L$ is the initial location,*
- *C is a finite set of clocks,*
- *V is a set of data variables,*
- *$\Sigma = (D \times \{?, !\}) \cup \{\tau\})$ is a finite set of messages where D is a set of message names and τ is the empty message,*
- *$R \subseteq \mathcal{R}$ is the set of side effects, $\epsilon \in R$ is the empty side effect,*
- *$E \subseteq L \times \mathcal{B}(C) \times \Sigma \times R \times 2^C \times D_u \times L$ is the set of edges where*
  - *$\varphi \in \mathcal{B}(C)$ are the time and variable guards,*
  - *$\lambda \in 2^C$ are the clock resets,*
  - *$d \in D_u$ is a set of data variable assignments where d can be any expression of "$v := exp$", "$out := v$", "$v := in$", with $v \in V$, and* exp *is an expression which uses arithmetic operations and evaluates to an integer,*
- *$I : L \to \mathcal{B}(C)$ assigns clock constraints to locations, the invariants.*

We write $l \xrightarrow{\varphi,a,r,\lambda,d} l'$ when $(l, \varphi, a, r, \lambda, d, l') \in E$.

A constraint B is a conjunctive formula of atomic constraints of the form $x \sim$ exp *for $x \in C \cup V$, $\sim \in \{\leq, <, =, >, \geq\}$. We use $\mathcal{B}(C)$ to denote the set of constraints over the set of clocks C.*

A timed automaton specifies time guards, clock resets, and invariants based on clocks. A time guard is a clock constraint that restricts the execution of a transition to a specific time interval. A clock reset sets the value of a clock back to zero while a transition is fired. Invariants are clock constraints associated

with locations that forbid that a timed automaton stays in a location when the clock values exceed the values of the invariant.

Variable assignments are modeled by strings that may be of three different forms. "$v := \exp''$ assigns the result of the expression exp to the variable v. "$out := v''$ models the sending of the variable v, and $v := in$ the receiving of v.

Figure 3.3 shows the simplified timed automaton that models the behavior of the component instance sa:Sanity of Figure 3.1. sa:Sanity checks the signal of dr:DSensor for plausible values. If a value deviates too much from the values, which have been measured directly before, it sends a reconfiguration proposal to st:Strategy.

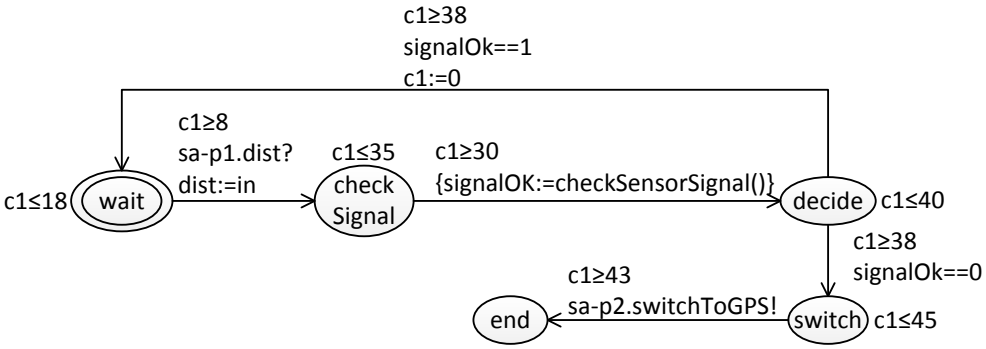


Figure 3.3: Timed automaton specifying the behavior of the component instance sa:Sanity

The timed automaton of the component instance sa:Sanity consists of five locations and five transitions connecting the locations. The invariant $c1 \leq 18$ of location wait specifies that wait may only be active while the value of c1 is less than or equal to 18. Accordingly, the time guard $c1 \geq 8$ restricts the firing of the transition from wait to checkSignal to a value greater or equal to eight. The time intervals are interpreted with respect to the values of clock c1 and not with respect to the global time that has passed since the system was started. The reset at the transition from decide to wait sets c1 back to zero.

A transition may carry a message symbol of Σ that specifies input messages and output messages of the timed automaton. Input messages are denoted by ?, output messages by !.

Remark 3.1 *In order to only exchange messages between timed automata of components and connectors that are connected in the deployment diagram, we relate messages to ports of component instances. We therefore use messages of the form k-p.m. k specifies the component instance, p the port, and m the message that is transmitted. For component types, we use messages of the form p.m and omit the name of the component type. If the message is used for internal component communication, the prefix k-p is omitted completely.*

In the timed automaton of sa:Sanity, the signal of the distance sensor dr:DistSensor is modeled by the synchronization sa-p1.dist? and the assignment dist:=in. The

prefix sa-p1 specifies that the message is received via port p1 of sa:Sanity. dist:=in denotes that the variable dist is received. The data is passed from a timed automaton which fires a transition with the message sa-p1.dist! and the assignment out:=dist that writes the distance value to the variable dist.

The side effect {signalOK:=checkSensorSignal()} executes the plausibility check. If the sensor signal is plausible, the side effect returns one and assigns it to the variable signalOK. Otherwise, the side effect returns zero. If the signal of the distance senor is plausible ($signalOK == 1$), the timed automaton returns to location wait. Otherwise, the reconfiguration proposal sa-p2.switchToGPS is sent to strt:Strategy via port p2.

Figure 3.4 shows the timed automaton that models the behavior of the component instance st:Strategy. If st:Strategy receives strt-p1.switchToGPS from sa:Sanity at any time, it triggers the side effect DiscDistSensor(). This side effect is specified by the self-healing operation, which is depicted in Figure 3.2. The self-healing operation disconnects dr:DSensor and ds:DSelect from the system.

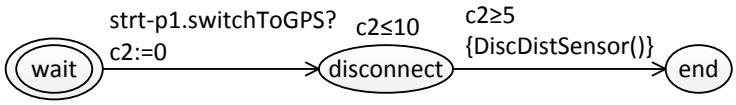


Figure 3.4: Timed automaton specifying the behavior of the component instance st:Strategy

Figure 3.5 shows the timed automaton that models the connector between sa:Sanity and st:Strategy. The timed automaton receives the synchronization sa-p1.switchToGPS at any time. When the synchronization is received, the clock cc1 is reset. After five to six time units the synchronization strt-p2.switchToGPS is forwarded to st:Strategy.

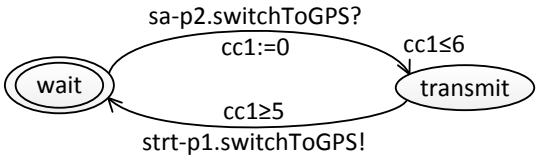


Figure 3.5: Timed automaton specifying the behavior of the connector between sa:Sanity and st:Strategy

Timed automata interact with each other using a joint set of messages. Such a set of interacting timed automata is referred to as a *network of timed automata* (NTA) [BY03]. We extend the definition by a set of shared variables to model the passing of parameters between timed automata.

Definition 3.3.2 (Network of Timed Automata)
We define $\mathcal{A} = \{A_1, \ldots, A_n\}$ *with* $n \in \mathbb{N}, n \geq 2$ *a* network of timed automata *(NTA).*

For all $A_i = (L_i, l_{0i}, C_i, V, \Sigma, R_i, E_i, I_i)$, $A_j = (L_j, l_{0j}, C_j, V, \Sigma, R_j, E_j, I_j) \in \mathcal{A}$ *with* $i, j \in \{1, \ldots, n\}, i \neq j$, $L_i \cap L_j = \varnothing$, $C_i \cap C_j = \varnothing$.

The timed automata of an NTA share a common alphabet Σ and a set of shared variables V. The timed automata of Figures 3.3, 3.4, and 3.5 build an NTA using the joint set of messages {sa-p2.switchToGPS, strt-p1.switchToGPS}.

The real-time statechart of a component type is transformed into an NTA that contains one timed automaton for each port of the component and one timed automaton specifying the internal behavior of the component. We denote this NTA as NTAC.

Definition 3.3.3 (NTAC)
We define NTAC(k) $= \{A_1, ...A_n\}$ *as the network of timed automata that specifies the behavior of the component type k. Without loss of generality, we define* A_1 *as the timed automaton that specifies the internal behavior of k.* $\{A_2, ..., A_n\}$ *specify the behaviors of the port types of k.*

We compute the propagation times of failures based on the reachable behavior of the system. The reachable behavior contains all runtime states that an NTA may visit during its execution. States of an NTA consist of the active locations of all timed automata in the network, a clock value assignment that assigns a value to each clock, and an assignment of data variables to values. Since clock values are real numbers, there exist infinitely many clock value assignments and, thus, infinitely many states. The problem is solved by using clock zones.

Definition 3.3.4 (Clock Zone)
Let C *be a set of clocks and* $B \subseteq \mathcal{B}(C)$. *We define a* clock zone $\hbar$ *by*

$$\hbar = \bigwedge_{b \in B} b.$$

If a clock zone $\hbar$ *contains any two of the constraints* $(a \sim_1 n_1)$, $(a \sim_2 n_2)$, $(a - b \sim_1 n_3)$, *and* $(b - a \sim_1 n_4)$ *for* $\sim_1 \in \{<, \leq\}$, $\sim_2 \in \{>, \geq\}$, $a, b \in C$, *and* $n_1, n_2, n_3, n_4 \in \mathbb{N}$, *we call it* normalized clock zone *[BY03].*

We denote the set of all clock zones by $\mathcal{H}$ *and the function* $\rho : \mathcal{H} \to 2^C$ *that returns the set of clocks* C *of a clock zone* $\hbar$. $\perp$ *is the empty clock zone.*

A clock zone h is a set of clock interpretations described by a conjunction of clock constraints each of which puts a lower or upper bound on a clock or on

the difference of two clocks. If C has k clocks, then h represents a convex set in the k-dimensional Euclidean space. [Alu99].

We define the reachable behavior of an NTA by means of zone graphs. Our definition of zone graphs is based on the definition of Bengtsson and Wang [BY03].

Definition 3.3.5 (Zone Graph)
Given is an NTA $\mathcal{A} = \{A_1, ..., A_n\}$ *with* $A_i = (L_i, l_{0_i}, C_i, V_i, \Sigma, R, E_i, I_i)$, $i \in \mathbb{N}$. *Its reachable state space is given by a* zone graph $Z = (S, s_0, \Sigma', T, V, \mu, \nu)$ *where*

- S *is the set of states,*
- s_0 *is the initial state,*
- Σ' *is the set of transition labels, and*
- $T \subseteq S \times \Sigma' \times S$ *is the set of transitions.*
- V *is a set of positive integer variables,*
- $\mu : V \to \mathbb{N}$, *and*
- $\nu : S \to 2^{\mathbb{N}^{|V|}}$.

States are tuples $(l, k, \hbar)$ *where*

- l *is a location vector,*
- $k \subseteq \mathbb{N}^{|V|}$ *is a set of integer vectors, and*
- $\hbar$ *is a normalized clock zone.*

Let l_i *denote the* i^{th} *element of the location vector* l *representing the active location of* A_i *and* $l[l'_i/l_i]$ *the vector* l *with* l_i *being substituted with* l'_i. *In* $s_0 = (l_{init}, k_{init}, \hbar_{init})$, *for all* A_i *and all clocks* $c_{init,j} \in \rho(\hbar_{init})$ *have value 0.*

Let $x(r)$ *denote the execution of a side effect* $r \in R$.

We define the set of transition labels *by* $\Sigma' = \{\delta\} \cup \{(i, \tau) | i \in \{1, \ldots, n\}\} \cup \Sigma'_{act}$ *with* $\Sigma'_{act} = \{((i, a?), (j, a!)) | i \in \{1, \ldots, n\}, j \in \{1, \ldots, n\}, a \in \Sigma\}$.

Let $e_j = (l_j, \varphi_j, a_j, r_j, \lambda_j, d_j, l'_j) \in E_j$ *and* $e_m = (l_m, \varphi_m, a_m, r_m, \lambda_m, d_m, l'_m) \in E_m$ *with* $j \neq m$. *The transitions of the zone graph* Z *are defined by the rules:*

1. $(l, k, \hbar) \xrightarrow{\delta} (l, k, \hbar{+}b)$ *if* $\hbar \in I(l)$ *and* $(\hbar{+}b) \in I(l)$, *where* $I(l) = \bigwedge_{l_i \in l} I(l_i)$ *and* $b \in \mathbb{R}^+$
2. $(l, k, \hbar) \xrightarrow{(j,\tau)} (l[l'_j/l_j], k, \hbar')$ *if* $l_j \xrightarrow{\varphi_j, \tau, \epsilon, \lambda_j, \varnothing} l'_j$, $\hbar' = ((\hbar \wedge \varphi_j)[\lambda_j \mapsto 0]) \wedge I(l[l'_j/l_j])$
3. $(l, k, \hbar) \xrightarrow{(j,r)} (l[l'_j/l_j], k, \hbar')$ *if* $l_j \xrightarrow{\varphi_j, \tau, r, \lambda_j, \varnothing} l'_j$, $\hbar' = ((\hbar \wedge \varphi_j)[\lambda_j \mapsto 0]) \wedge I(l[l'_j/l_j] \wedge x(r))$
4. $(l, k, \hbar) \xrightarrow{((j,a?,r),(m,a!,s))} (l[l'_j/l_j][l'_m/l_m], k', \hbar')$ *where* $r, q \in R$ *if*

a) $l_j \xrightarrow{\varphi_j, a?, r, \lambda_j, d_j} l'_j$, $l_m \xrightarrow{\varphi_m, a!, q, \lambda_m, d_m} l'_m$, *with* $(r = \epsilon \wedge x(q))$ *xor* $(q = \epsilon \wedge x(r))$,

b) $d_j =$"$v := in$", $d_m =$"$out := v$", $v \in V$,

c) $d_j =$"$v_j := \exp_j$", $d_m =$"$v_m := \exp_m$", $v_j, v_m \in V$, $\exp_j$ *and* $\exp_m$ *are expressions which use arithmetic operations and evaluate to integers, or*

d) $\hbar' = ((\hbar \wedge \varphi_j \wedge \varphi_m)[\lambda_j \cup \lambda_m \mapsto 0]) \wedge I(l[l'_j/l_j][l'_m/l_m])$.

States of the zone graph are tuples $(l, k, \hbar)$. l stores the active locations for each timed automaton of the NTA. Each integer set of k stores the evaluations of a variable that are possible for the variable in this state. The normalized clock zone $\hbar$ contains all possible clock interpretations of the state.

We compute the zone graph of an NTA by symbolic execution using the approach of Heinzemann et al. [HSE10].

In addition to the semantics defined by Bengtsson and Wang [BY03], we allow shared positive integer variables as introduced by Behrmann et al. [BDL+06]. Therefore, we add a set of positive integer variables V and the functions μ and ν which map variables to sets of positive integers and states to vectors of sets of positive integers. Values are transmitted in one direction per transition and without conditions. The sender writes data to the shared variable v using the string "$out := v$" and performs a send message $a!$. The receiver receives the co-message $a?$ and reads the variable v by "$v := in$". The receiver can access the transmitted data in the same transition, because the send message $a!$ is always evaluated before the receive message $a?$.

Side effects of transitions in timed automata are executed when the transitions fire.

Figure 3.6 shows the zone graph of the NTA that is built by the timed automata of the component instances sa:Sanity (cf. Figure 3.3) and strt:Strategy (cf. Figure 3.4), and the connector between them (cf. Figure 3.5). States are labeled by clock zones. Transitions are labeled with δ (time passes), τ (silent transition), or the messages, which are transmitted. The syntax of messages is the same as in timed automata.

3.4 Timed Failure Propagation Graphs

We use timed failure propagation models called *timed failure propagation graphs* (TFPG) to analyze the failure propagation over time. TFPGs, like common failure propagation models [VGRH81], define a cause-consequence-relation between failures. In particular, TFPGs include *propagation time intervals* that specify minimum and maximum propagation times between failures.

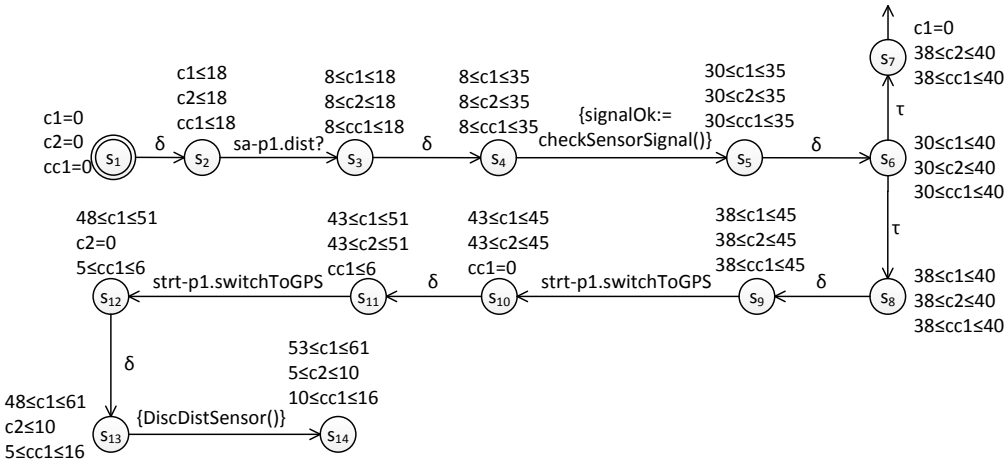


Figure 3.6: Zone graph of the NTA of Figures 3.3 to 3.5

The benefit of TFPGs is their minimality concerning the information needed for the analysis of failure propagation times. This analysis actually requires taking the reachable behavior of the whole system into account. The reachable behavior of a system comprises the complete data and control flow. However, for analyzing the propagation times of failures, we only need to take the relations between failures at the ports of components into account. We therefore abstract from the system behavior by using TFPGs.

For the specification of failure propagation, we follow the terminology of Avizienis et al. [ALRL04] (cf. Section 2.3). *Failures* are the externally visible deviation from the component's behavior. They are associated with ports where the component instances interact with their environment. *Errors* are the manifestation of a *fault* in the state of a component, whereas a fault is the cause of an error. Errors are restricted to the internals of hardware nodes.

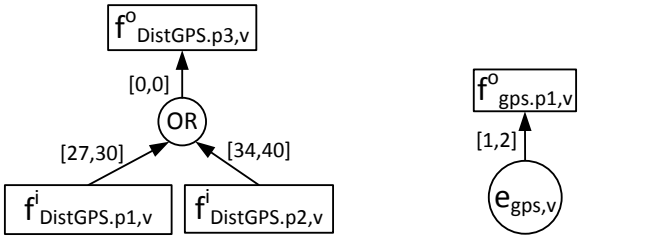
Errors and failures are classified using a failure classification like the one by Fenelon et al.[FMNP94] (cf. Section 2.3). We distinguish the failure classes *service*, *value*, *early timing*, and *late timing*. A value failure specifies that a value deviates from a correct value, e.g., an erroneous message at the port of a component. A service failure specifies that no value is present at all, e.g., a component crashed and does not output any values. A timing failure specifies that a message has been delivered outside of a defined time interval, e.g., too late or too early.

TFPGs are generated automatically from the real-time statecharts of component types. The generation of TFPGs is described in Chapter 4. We generate a TFPG for each component type of the system. TFPGs are instantiated in the deployment diagram at the same time when the component types are instantiated. The connections between the TFPGs of the component instances and between the TFPGs of component instances and hardware nodes are created according to the connectors between the component instances and hardware nodes. The delays of connectors are computed from their real-time statecharts.

By generating TFPGs for component types instead of component instances, we decrease the effort of the TFPG generation for a system: We generate only one TFPG for the component type and use it for each instance of that component type.

Our TFPG generation can be applied to hardware nodes, as well. Therefore, the developer needs to specify the behavior of the hardware nodes by a real-time statechart or timed automaton.

In TFPGs, failures are represented by rectangles that are labeled with the according failure variable (cf. Figure 3.7). Errors are drawn as circles that are labeled with the according error variable. Operators are depicted by circles labeled with the according logical operator. In the remainder, we write "AND-node" for nodes labeled with & and "OR-node" for nodes labeled with OR. Edges are labeled with propagation time intervals.



(a) Component type DistGPS (b) Hardware node gps:GPS

Figure 3.7: TFPGs

Remark 3.2 *For a component k, a port p, an error or failure class c, and a direction d, error and failure variables are named according to the following scheme: $e_{k,c}$ and $f^d_{k.p,c}$ with $d \in \{\mathrm{i}, \mathrm{o}\}$ and $fc \in \{v, s, e, l\}$. i and o specify the direction of failures – i stands for incoming and o for outgoing. For error and failure classes, we use the following symbols: v for value, s for service, e for early timing, and l for late timing. For a deployment, we instantiate failure variables by instantiating components. The notation described above holds for component instances, port instances, hardware nodes, and hardware ports analogously.*

The TFPG of Figure 3.7(a) specifies a part of the failure propagation of the component type DistGPS of Figure 3.1. The TFPG relates the outgoing value failure $f^o_{DistGPS.p3,v}$ at port p3 of DistGPS to the incoming value failures $f^i_{DistGPS.p1,v}$ and $f^i_{DistGPS.p2,v}$ at ports p1 and p2. The operator OR specifies that $f^o_{DistGPS.p3,v}$ occurs if either of the incoming failures occurs.

The propagation time interval at the edge from $f^i_{DistGPS.p1,v}$ to the OR-node in Figure 3.7(a) specifies that a value failure needs at minimum 27 and at maximum 30 time units to propagate from port p1 to the OR-node. The edge originating from the OR-node has a propagation time interval of $[0, 0]$. This means between the OR-node and port p3, failures propagate in zero time. These time intervals are introduced by the automatic generation as we will explain in

Chapter 4. Thus, a failure needs between 27 and 30 time units to propagate from port p1 to port p3.

Figure 3.7(b) shows the TFPG of the hardware node gps:GPS. The value error $e_{gps,v}$ which may occur in this hardware node is represented by a circle. The TFPG specifies that $e_{gps,v}$ will propagate to the value failure $f^o_{gps.p1,v}$ at port p1 within one to two time units.

Figure 3.8 shows the TFPG of the deployment diagram of the speed control subsystem of Figure 3.1. The component type DistGPS has been instantiated by the component instance dg:DistGPS. The incoming and outgoing failures of the TFPG of dg:DistGPS are connected to failures of connected component instances according to the connectors of Figure 3.1. In deployment diagrams, edges in the TFPG are only created if the failure classes at the ports of the connected component instances are of the same failure class or if one failure class is a generalization of the other [GTS04]. For example, the outgoing value failure $f^o_{dg.p3,v}$ is connected to the incoming value failure $f^i_{ds.p1,v}$ of ds:DSelect, because dg:DistGPS and ds:DSelect are connected in the deployment diagram of Figure 3.1 and the failures are both of the failure class value. The edge is labeled with the propagation time interval $[5, 6]$ of the connector. This edge specifies that the propagation of a value failure from dg:DistGPS to ds:DSelect takes between five and six time units.

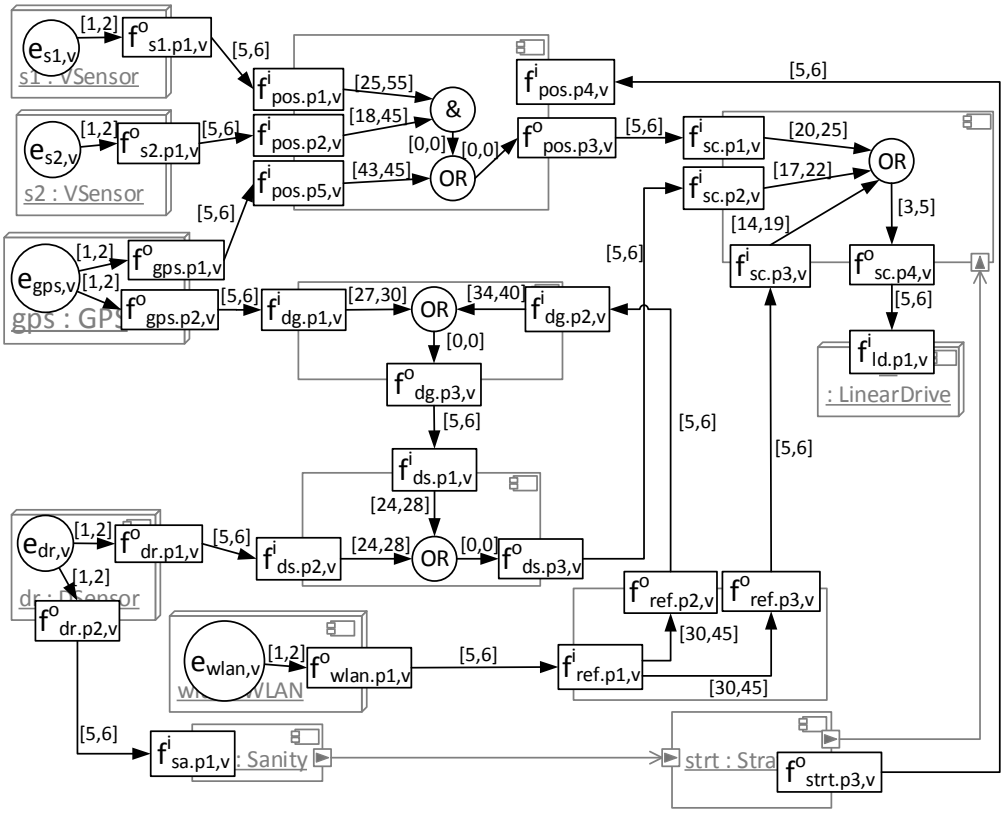


Figure 3.8: Deployment diagram with TFPG

The error causing the failure $f^o_{strt.p3,v}$ which originates from strt:Strategy is not shown here, because this failures is of no relevance in our example. The incoming failure $f^i_{pos.p4,v}$ which results from the latter outgoing failure $f^o_{strt.p3,v}$ is not connected to the outgoing failure $f^o_{pos.p3,v}$, because it does not affect it. As we will see in Chapter 4, $f^i_{pos.p4,v}$ causes the outgoing timing failure $f^o_{pos.p3,t}$ which is not part of the depicted TFPG. In the remainder of this thesis, we will omit $f^o_{strt.p3,v}$ and $f^i_{pos.p4,v}$ from the TFPG of the speed control subsystem, because they are of no relevance for the considered hazard wrong_speed.

3.4.1 Formalization

The definition of TFPGs requires defining which errors and failures may occur in the system. We formalize this by the error and failure specification.

Definition 3.4.1 (Error and Failure Specification, $\mathcal{V}$, $\overline{\mathcal{V}}$)
Let sys $= (V_{sys}, E_{sys}, s_{sys}, t_{sys})$ *be a component specification, with* $V_{sys} = K \cup H \cup P_K \cup P_H \cup L$ *with the component types* K*, the hardware nodes* H*, the port types* P_K*, the hardware ports* P_H*, and the connector types* L*.*

We denote an error variable *by* $e = (h, c_e)$ *with* $h \in H$ *and the error class* c_e*.*

We denote a failure variable *by* $f = (k, p, c_f, d)$ *with* $k \in K \cup H$*, the port type or hardware port* $p \in P_K \cup P_H$*, the failure class* c_f*, and the direction* d*.*

We denote the set of error variables *by* $\mathcal{E}$ *and the* set of failure variables *by* $\mathcal{F}$*.*

We define the error and failure specification $\mathcal{V}(\text{sys}) = (\text{sys}, \mathcal{E}, \mathcal{F}, f_\mathcal{E}, f_\mathcal{F})$ *with*

- $\mathcal{E}$ *the set of error variables,*
- $\mathcal{F}$ *the set of failure variables,*
- $f_\mathcal{E} : \mathcal{E} \to H$ *(bijective) maps error variables to hardware nodes, and*
- $f_\mathcal{F} : \mathcal{F} \to P_K \cup P_H$ *(bijective) maps failure variables to port types and hardware ports.*

Let $w = (G_{conf} = (V_{conf}, E_{conf}, s_{conf}, t_{conf}), type)$ *be a deployment diagram over* sys *with* $V_{conf} = \overline{K} \cup \overline{H} \cup \overline{P}_K \cup \overline{P}_H$*.*

We define the error and failure specification $\overline{\mathcal{V}}(w) = (w, \overline{\mathcal{E}}, \overline{\mathcal{F}}, \overline{f}_\mathcal{E}, \overline{f}_\mathcal{F})$ *with*

- $\overline{\mathcal{E}}$ *the set of error variables,*
- $\overline{\mathcal{F}}$ *the set of failure variables,*
- $\overline{f}_\mathcal{E} : \overline{\mathcal{E}} \to \overline{H}$ *(bijective) maps error variables to hardware nodes, and*
- $\overline{f}_\mathcal{F} : \overline{\mathcal{F}} \to \overline{P}_K \cup \overline{P}_H$ *(bijective) maps failure variables to port instances and hardware ports.*

$\mathcal{V}$ specifies the error and failure variables on type level and $\overline{\mathcal{V}}$ defines the error and failure variables on instance level. $\overline{\mathcal{V}}$ is bijective, because hardware nodes, hardware ports, and component instances are unique.

The component specification of Def. 3.2.1 and the error and failure specification of Def. 3.4.1 are used to define the TFPG syntax.

Definition 3.4.2 (Timed Failure Propagation Graph)
Let $\mathcal{V} = (\text{sys}, \mathcal{E}, \mathcal{F}, f_{\mathcal{E}}, f_{\mathcal{F}})$ be an error and failure specification.

We define $O = \{AND, OR\}$ as the set of operators.

We then define the timed failure propagation graph *(TFPG) $G = (V, E, f_s, f_t, I, l, \iota, \eta)$ as a labeled graph (cf. [EEPT06]) over $\mathcal{V}$ where*

- *V is the set of* nodes*,*
- *$E \subseteq V \times V$ is the set of* edges*,*
- *$f_s, f_t : E \to V$ are the* source *and* target *functions,*
- *$I = \{[t_{min}, t_{max}] \mid t_{min}, t_{max} \in \mathbb{Q}_{\geq 0},\ t_{min} \leq t_{max}\}$ is the set of* propagation time intervals*,*
- *$l : V \to \mathcal{E} \cup \mathcal{F} \cup O$ is the* node labeling *function,*
- *$\iota : E \to I$ is the* edge labeling *function, and*
- *$\eta : V \to \{\text{active}, \text{inactive}\}$.*

We define $\delta^+(v) = |\{e \in E \mid f_s(e) = v\}|$ as the outdegree *and $\delta^-(v) = |\{e \in E \mid f_t(e) = v\}|$ as the* indegree *of a node $v \in V$. Let $V_0 = \{v \in V \mid \delta^-(v) = 0\}$. Then $\forall v \in V_0 : l(v) \in \mathcal{E} \cup \mathcal{F}$ and $\forall v \in V \setminus V_0 : l(v) \in \mathcal{F} \cup O$ hold.*

We avoid real numbers as interval bounds of propagation time intervals to enable mapping to time Petri nets. This mapping will be introduced in Definition 3.4.4.

Nodes in the TFPG are set to *active* or *inactive* to allow for tracing failures through the system. Errors and failures, which have occurred in the system, are represented by active nodes. All other nodes are inactive.

All nodes with indegree zero are labeled with error variables. All other nodes are labeled with either a failure variable or a logical operator ≥ 1 or &.

We define the semantics of TFPGs by time Petri nets (TPN) [CR05]. TPNs are marked Petri nets [Reu90] with a time extension. Below, we formalize TPNs as presented in [CR05]. We assume the same semantics.

Definition 3.4.3 (Time Petri Net (TPN) [CR05])
A timed Petri net *(TPN)* $\mathcal{T}$ *is a tuple* $(P, T, \bullet(.), (.)\bullet, M_0, (\alpha, \beta))$ *where*

- $P = \{p_1, p_2, ..., p_{|P|}\}$ *is a finite* set of places,
- $T = \{t_1, t_2, ..., t_{|T|}\}$ *is a finite* set of transitions,
- $\bullet(.) : T \to \mathbb{N}_0^{|P|}$ *is the* backward incidence mapping,
- $(.)\bullet : T \to \mathbb{N}_0^{|P|}$ *is the* forward incidence mapping,
- $M_0 \in \mathbb{N}_0^{|P|}$ *is the initial marking,*
- $\alpha \in \mathbb{Q}_{\geq 0}^{|T|}$ *and* $\beta \in (\mathbb{Q}_{\geq 0} \cup \{\infty\})^{|T|}$ *are the* earliest *and the* latest firing time mappings.

In this work, we consider TPNs whose transitions are labeled with time intervals [CR05]. The time interval of each transition specifies the earliest and latest firing time relative to the internal clock of the transition. The internal clock starts with zero at the time when the transition is activated. The transition can only fire if its clock has a value that is within the transition's time interval.

We use the morphism defined in Definition 3.4.4 below to map a TFPG to a TPN.

Definition 3.4.4 (Morphism from TFPG to TPN)
We define a graph morphism $\mu : G \mapsto \mathcal{T}$ *from a TFPG* $G = (V, E, f_s, f_t, I, l, \iota, \eta)$ *to a TPN* $\mathcal{T} = (P, T, \bullet(.), (.)\bullet, M_0, (\alpha, \beta))$ *as a tuple* $\mu = (\mu_V, \mu_E, \mu_I)$ *where*

- $\mu_V : V \to P$ *(bijective),*
- $\mu_E : E \to T$ *(bijective),*
- $\mu_I : I \to \mathbb{Q}_{\geq 0}^{|T|} \times \mathbb{Q}_{\geq 0}^{|T|}$ *(bijective) maps the propagation time intervals to the earliest and latest firing time mappings.*

with

- *For all* $t \in T$ *the backward incidence mapping is described by* ${}^\bullet(t) = (v_1, ..., v_{|P|})$*, where*

$$v_i = \begin{cases} x & \mu_V^{-1}(p_i) = f_s(\mu_E^{-1}(t)) \\ 0 & else \end{cases}$$

with

$$x = \begin{cases} \delta^-(f_s(\mu_E^{-1}(t))) & l(f_s(\mu_E^{-1}(t))) = \& \\ 1 & else \end{cases}$$

- *For all* $t \in T$ *the forward incidence mapping is described by* $(t)^\bullet = (v_1, ..., v_{|P|})$*, where*

$$v_i = \begin{cases} 1 & \mu_V^{-1}(p_i) = f_t(\mu_E^{-1}(t)) \\ 0 & else \end{cases}$$

- $M_0 = (m_1, ..., m_{|P|})$*, where*

$$m_i = \begin{cases} 1 & \eta(\mu^{-1}(p_i)) = \text{active} \\ 0 & else \end{cases}$$

- $\alpha = (\alpha_1, ..., \alpha_{|T|})$ *where* $\alpha_i = \min(\iota(\mu_E^{-1}(t_i)))$
- $\beta = (\beta_1, ..., \beta_{|T|})$ *where* $\beta_i = \max(\iota(\mu_E^{-1}(t_i)))$

Nodes of the TFPG are mapped to places and edges to transitions. Propagation time intervals become firing time mappings. The minimum propagation time is mapped to the earliest firing time mapping and the maximum propagation time to the latest firing time mapping.

The backward and forward incidence mappings of the TPN are represented by vectors. There exists one vector for the backward and forward incidence mapping of each transition, respectively. The size of each vector is equal to the number of places in the TPN. The entries of the backward incidence mapping specify the number of tokens needed to activate the transition. The entries of the forward incidence mapping specify the number of tokens that move from the transition into the subsequent place.

To model the propagation over logical nodes in the TFPG correctly, we need to restrict the propagation via edges leaving AND-nodes in the TFPG. This is achieved by setting the backward incidence mapping of transitions originating from an AND-node to the sum of edges pointing at the AND-node. The backward incidence mapping of all other transitions is one.

Active nodes in the TFPG are mapped to the initial marking. Places that correspond to active nodes in the TFPG are assigned one token. All other place do not contain tokens.

Figure 3.9 shows the TPN of the TFPG of the component type DistGPS of Figure 3.7(a). Places are represented by circles and transitions by rectangles. Arrows connect places and transitions and indicate the direction of the flow of tokens. Places are labeled by logical operators or error or failure variables. Transitions are labeled by time intervals that define the earliest and latest firing time of the transitions.

In order to analyze which error and failure variables are active during a specific time interval, we define the state of a TFPG. TFPG-states are based on TPN-

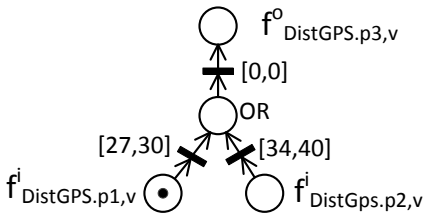


Figure 3.9: TPN of the TFPG of Figure 3.7(a)

states. A state of a TPN specifies a marking for a period of time. This period of time is represented by a normalized clock zone.

Definition 3.4.5 (State of a TPN, State of a TFPG)
A state $s(\mathcal{T}) = (m, \hbar)$ *of a TPN* $\mathcal{T} = (P, T, {}^\bullet(.), (.)^\bullet, M_0, (\alpha, \beta))$ *is defined by a marking* $m \in \mathbb{N}^{|P|}$ *and a clock zone* $\hbar$ *[CR05].*

A state $q(G)$ *of a TFPG* $G = (V, E, f_s, f_t, I, l, \iota, \eta)$ *over the error and failure specification* $\mathcal{V} = (sys, \mathcal{E}, \mathcal{F}, f_\mathcal{E}, f_\mathcal{F})$ *is defined by a set of active error and failure variables* $q(G) \subseteq \mathcal{E} \cup \mathcal{F}$*. Let* $\mathcal{T}$ *be the underlying TPN of* G *and* $\mathcal{T} = \mu(G)$*,* $\mu = (\mu_V, \mu_E, \mu_I)$ *. Let* $M = (m_1, ..., m_{|P|}) \in \mathbb{N}^{|P|}$ *be a marking in* $\mathcal{T}$ *for the clock zone* $\hbar$*. Then*

$$q(G) = \left(\bigcup_{i=1}^{|P|} \{ l(\mu_V^{-1}(p_i)) \mid m_i > 0 \} \right) \cap (\mathcal{E} \cup \mathcal{F}) \,.$$

$q(G)$ collects all active error and failure variables which are represented by a token in the underlying TPN of the TFPG G during the time span specified by the clock zone $\hbar$.

A state of a TFPG may represent more than one marking of a TPN.

Figure 3.10 shows the TPN of the TFPG of the speed control subsystem Figure 3.8. It illustrates the state of the TPN for the period $[38, 45]$. The state is a marking for the clock zone $[38, 45]$. It contains the elements that correspond to the places labeled with $f^i_{sc.p1,v}$ and $f^i_{ds.p1,v}$. The places initially marked with a token were $e_{s1,v}$, $e_{s2,v}$, and $e_{gps,v}$.

The definition of the semantics of TFPGs allows to apply the TPN reachability analysis of Cassez and Roux [CR05] on TFPGs. This, in turn, allows for analyzing how far failures may propagate during a specific time interval.

3.4.2 Adjusting the Propagation Time Intervals of TFPGs

Failures may not always have an immediate impact on the system. If, for example, a sensor delivers a single peak value, this peak value may be corrected

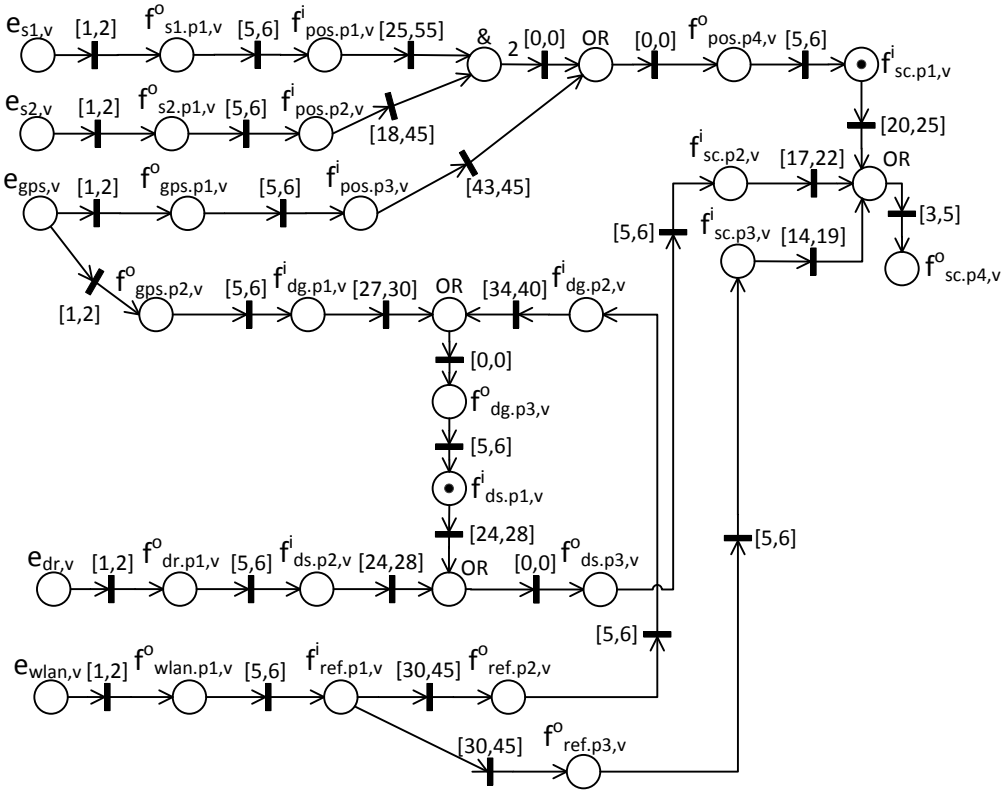


Figure 3.10: TPN of the TFPG of Figure 3.8

by smoothing the signal by a low pass filter [AH99]. Only if the deviation from the correct signal occurs in a several consecutive signals, this may have an adverse effect on the system. This fact needs to be taken into account in AShOp because there is more time for the system to react by self-healing if a controller tolerates a certain amount of wrong values or omissions. We therefore integrate this time into the TFPG and call this time *tolerance time*

Each controller reads signals periodically. Each period between two signals has a specific duration. The number of tolerable wrong values specifies how many periods of wrong values are tolerable. We thus compute the tolerance time by the product of the tolerated wrong values and the period with which the controller reads the signals.

The speed controller sc:SpeedCtrl of the speed control subsystem of Figure 3.1 tolerates a maximum of five consecutive wrong distance values. From the specification of the controller, we know that the speed controller receives a distance signal every 24 time units. Thus, the tolerance time is $5 \cdot 24 = 120$.

The tolerance time is added to the propagation time interval of each error, which is a cause of the incoming failure of the controller. We therefore extend both values of the propagation time interval of all these errors by the tolerance time. Such in the TFPG, the error is masked by the sensor as long as it is tolerated by the controller. Only the propagation of the first non-tolerated

error is analyzed. To identify these errors, we track back all paths in the TFPG from the controller to the errors by depth-first-search.

The speed sensor of our example receives the distance values via port p2. This means, errors in the distance measurement cause the incoming failure $f^i_{sc.p2,v}$. The causes for this failure are the errors $e_{gps,v}$, $e_{dr,v}$, and $e_{wlan,v}$. Consequently, the propagation time intervals at the edges from $e_{gps,v}$ to $f^o_{gps.p2,v}$, from $e_{dr,v}$ to $f^o_{dr.p1,v}$, and from $e_{wlan,v}$ to $f^o_{wlan.p1,v}$ are increased by the tolerance time from $[1, 2]$ to $[121, 122]$. Figure 3.11 shows the resulting TFPG.

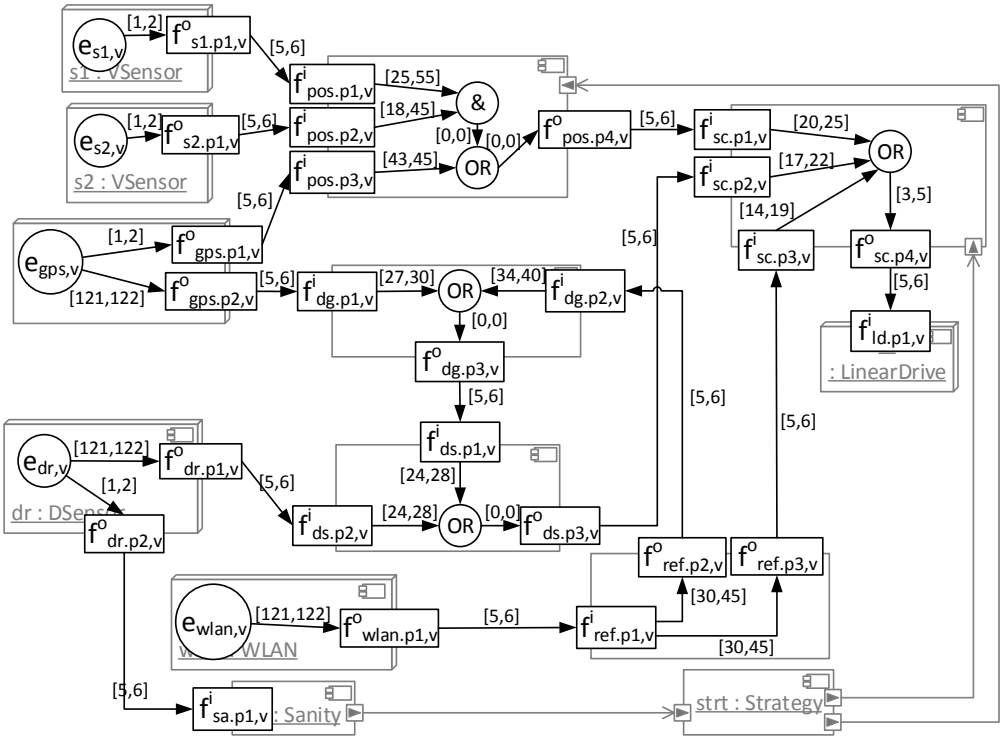


Figure 3.11: TFPG of Figure 3.8 with integrated tolerance time

3.4.3 Component-based Hazard Analysis Using TFPGs

Before the developer specifies the self-healing operations, he needs to know which hazards have to be reduced and which errors cause these hazards. This is computed by the component-based hazard analysis of Giese et al. [GT06] (cf. Section 2.4.2). It operates on Boolean formulas, which may be computed from TFPGs. For this, all nodes of the TFPG with an outdegree greater than one have to be divided into subnodes with outdegree one. Furthermore, all paths consisting solely of failure variables are replaced by edges. Propagation time intervals are removed. The result is a syntax tree that can be mapped to the corresponding Boolean formula. For example, the Boolean formula of the

TFPG of the component type DistGPS of Figure 3.7(a) is

$$f^o_{DistGPS.p3,v} \Leftrightarrow f^i_{DistGPS.p1,v} \vee f^i_{DistGPS.p2,v}.$$

We use the TFPG of Figure 3.8 to compute the MCSs of the hazard wrong speed. The MCSs are $\{e_{s1,v}, e_{s2,v}\}$, $\{e_{gps,v}\}$, $\{e_{dr,v}\}$, and $\{e_{wlan,v}\}$. To compute the occurrence probability of the hazard, we assume that each of the errors occurs with a probability of 0.1. Then, the occurrence probability of the hazard wrong_speed is

$$\begin{aligned} & 1 - ((1 - p(e_{s1,v}) \cdot p(e_{s2,v}))(1 - p(e_{gps,v}))(1 - p(e_{dr,v}))(1 - p(e_{wlan,v}))) \\ =& 1 - ((1 - 0.01)(1 - 0.1)(1 - 0.1)(1 - 0.1)) \\ =& 0.27829 \end{aligned}$$

In our example, the threshold for the occurrence probability of the hazard wrong speed is 0.2. The computed occurrence probability of 0.27829 exceeds this threshold. Consequently, the developer needs to reduce the occurrence probability of this hazard.

3.5 Summary

In this chapter, we have presented a modeling formalism that allows for modeling the propagation of failures through the system and in particular allows for analyzing the propagation times of failures. The propagation times of failures are needed to analyze how far failures may propagate through the system within a specific time interval. In order to apply a formal analysis that computes how far failures may propagate, we defined a formal semantics of TFPGs by means of time Petri nets. TFPGs may also be used to compute minimal cut sets and hazard occurrence probabilities, for example by the hazard analysis of Giese et al. [GTS04, GT06]. In the next chapter, we show how TFPGs are generated from the real-time statecharts that specify the behavior of component types.

4 Generation of Timed Failure Propagation Graphs

In this chapter, we present our approach for the generation of TFPGs (cf. Section 3.4) from the real-time statecharts (cf. Section 2.2.2) of component types as published in [PHS13].

Existing approaches [LR98, KLFL11, MWP11] already generate fault trees from behavior models. The underlying idea of these approaches is to compare the reachable behavior of a component with the behavior of the component with injected failures. Based on this information, the construction of failure propagation models is guided by the rules introduced by Vesely et al. [VGRH81].

We adapt the idea of these approaches and extend it by the identification of failure classes and the computation of propagation times to generate TFPGs. Failure classes help to distinguish failures with different properties. These properties reflect how the failure affects the system. This distinction makes the analysis more precise. The propagation times are computed from the time constraints of our behavior models.

The input for our TFPG generation is the real-time statechart (cf. Section 2.2.2) that specifies the behavior of a component type. However, we define our approach based on timed automata (cf. Section 3.3), because they are well established in literature. Real-time statecharts can be transformed into networks of timed automata (NTA) (cf. Def. 3.3.2) as illustrated by Burmester et al. [BGHS04].

Figure 4.1 shows an overview of our TFPG generation. The input is the NTAC (cf. Definition 3.3.3), the NTA which specifies the behavior of a component type.

One realistic assumption of our approach is that each location of the NTAC that has outgoing edges must have an invariant that limits the time the NTAC is allowed to stay in this location. Otherwise, it would be possible to stay in a location infinitely long. This would result in an infinitely long propagation time, which is not defined for TFPGs (cf. Def. 3.4.2).

In order to construct the TFPGs, we first identify which incoming failures cause which outgoing failures of each component type. Each time, such a relation is identified, the propagation times between the related failures are computed and the relation is stored in a TFPG. This is repeated until all combinations of incoming failures have been evaluated. The construction of TFPGs from relations between incoming and outgoing failures will be explained in Section 4.2.

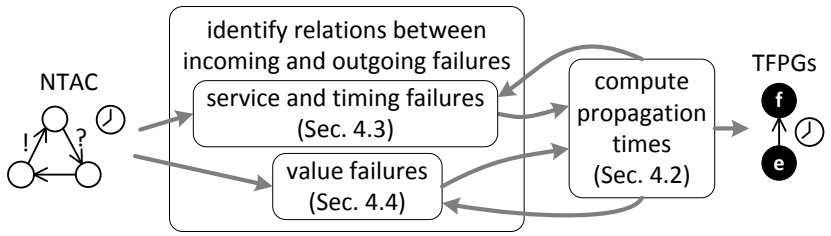


Figure 4.1: Generation of TFPGs from timed automata

We have to distinguish between the identification of relations between outgoing and incoming timing and service failures on the one hand and outgoing and incoming value failures on the other hand.

Service and timing failures change the control flow such that either no message is sent or a message is sent too early or too late. Causes for these deviations may be that either transitions, which should have been fired, could not be activated or other transitions with other time constraints have fired. Relations between incoming and outgoing timing and service failures are therefore identified by deviations in the control flow. In Section 4.2.1, we present a method which analyzes these deviations by comparing the reachable behavior (cf. Def. 3.3.5) of the NTAC with and without injected failures.

Value failures cannot be detected by deviations in the control flow, because even though the same transition is fired, the values of variables may differ. Consequently, we need to identify relations between incoming and outgoing value failures from the data flow. In Section 4.2.2, we explain how we use slicing on extended finite state machines [ACH+12] to identify these relations.

We generate TFPGs for component types. However, at type level, we do not know how instances of the component type will be connected in a deployment diagram. We consequently need to take all possible incoming and outgoing failures of the component type into account when generating TFPGs. Nevertheless, at instance level, AShOp only takes incoming failures into account that result from outgoing failures of connected component instances. This is because in deployment diagrams, edges in the TFPG are only created if the failure classes at the ports of the connected component instances are of the same failure class or if one failure class is a generalization of the other [GTS04]. Thus, incoming failures of a failure class, which are modeled in the TFPG of the component type but do not occur in the deployment diagram, are ignored.

If, however, the developer knows where the instances of a component type will be placed in the deployment diagram, he may specify incoming failure types to reduce the complexity of the generation. The outgoing failure types of a component are then determined by the generation.

4.1 Example

Figure 4.2 shows the simplified timed automaton that models the behavior of the component type PosCalc that was introduced in Figure 3.1. We will use this example to illustrate the generation of TFPGs. PosCalc reads the signals of the two speed sensors and computes the position of the RailCab on the track. Before, PosCalc receives a message from strt:Strategy about which drive mode is activated: either fast or slow. Depending on the drive mode, the position data is output in shorter or longer time intervals. The processing of the GPS-signal is omitted to keep the example simple.

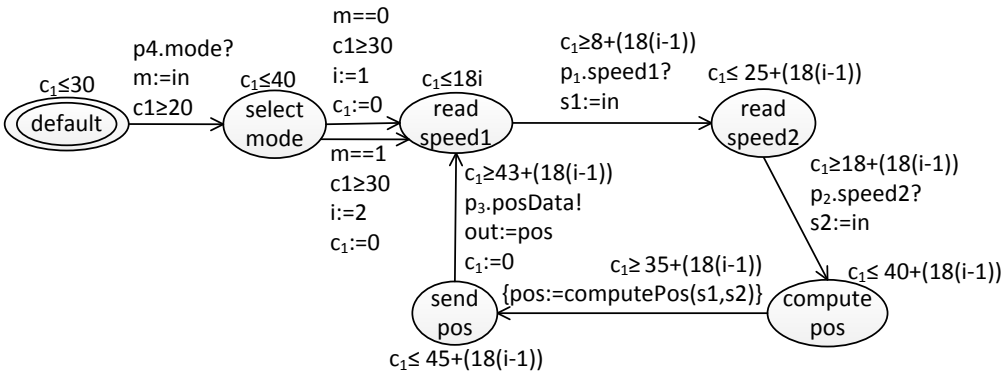


Figure 4.2: Timed automaton specifying the behavior of the component type PosCalc

The variable m defines the drive mode. It is received via port p4 with the message p4.mode?. Variable m affects the value of variable i which is set at the transitions from select_mode to read_speed1. Variable i affects the time guards at all transitions after read_speed1 and thus the frequency with which the position of the RailCab is updated. If m is set to zero, the higher transition from select_mode to read_speed1 fires and i is set to zero. If m is set to one, the lower transition fires and i is set to one. If i is one, the transitions will fire within shorter time intervals (fast drive). If i is two, the transitions will fire within longer time intervals (slow drive).

When the drive mode is set, PosCalc receives the variables s1 and s2 which contain the signals measured by the two speed senors. The position of the RailCab is computed by pos:=(computePos(s1,s2)) using the the speed sensor signals. Finally, the position is sent to the speed controller via variable pos.

4.2 Constructing TFPGs

The construction of TFPGs is based on the construction of fault trees as introduced by Vesely et al. [VGRH81]. We first identify logical connections between incoming failures and store them in a TFPG. Then, we compute propagation time intervals and add them to the TFPG.

The construction of logical connections is illustrated in Figure 4.3. For each failure class of a port, we create a separate TFPG. The top event f^o of this TFPG is a failure which represents all outgoing failures of a failure class. The basic events are the incoming failures $f^i_{11}...f^i_{nm}$. A combination of incoming failures, which causes an outgoing failure, is connected by an AND-operator, because all failures of this combination have to occur at the same time. All combinations of incoming failures that lead to an outgoing failure of the same failure class at the same port are connected by an OR-node, because the failure will occur if any of these combinations occurs.

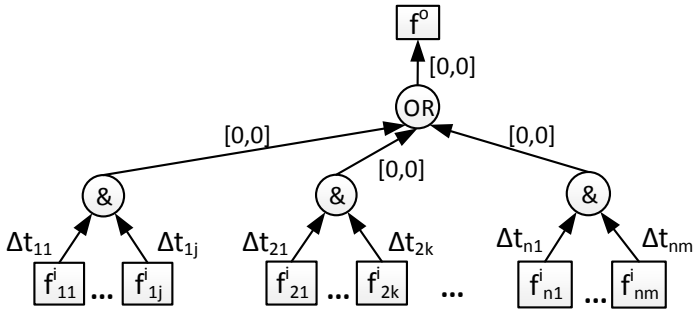


Figure 4.3: TFPG for the outgoing failures of one failure class

The propagation time intervals $\Delta t_{11}, ..., \Delta t_{nm}$ at the edges of the TFPG are computed from runtimes of paths in the reachable behavior of the NTAC. The computation of propagation times will be explained in Section 4.2.1.

4.2.1 Timing and Service Failures

The relations between incoming and outgoing timing and service failures are identified by deviations in the control flow. To provoke these deviations, the control flow is modified by injecting failures into the NTAC. Failures enter a component via faulty messages. Thus, for injecting a timing failure, a message is sent earlier or later than it is specified by the NTAC. For a service failure, a message that is expected by the NTAC is not sent at all. These modified messages may change the control flow of the NTAC. Outgoing timing and service failures are identified by deviations between the original and the modified control flow.

To analyze the control flow, we construct the reachable behavior of the NTAC for the case that no failures are injected (cf. Section 4.2.1). We call this behavior the *normal behavior* of the NTAC. We further construct a reachable behavior for each combination of incoming timing and service failures of the NTAC (cf. Section 4.2.1). This reachable behavior is the *failure behavior* of the NTAC. We compare both behaviors to identify relations between incoming and outgoing timing and service failures.

Computing the Normal Behavior

The behavior of a component depends on the messages, which are received via its ports. If the component is embedded in a system, these messages are sent by the component's environment, i.e., other component instances or hardware nodes. However, this environment does not exist for component types, because component types are independent of deployment diagrams. Still, the incoming and outgoing messages need to be sent and received for symbolic execution of the component type. We therefore model this environment by a set of timed automata as illustrated in Figure 4.4. We call this environment *component context*. The bold arrows illustrate the message flow between the NTAC and the component context. The component context and the NTAC build an NTA.

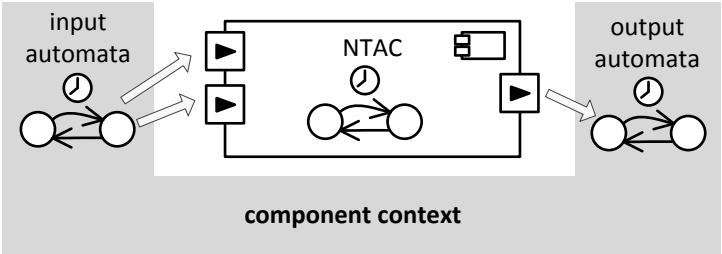


Figure 4.4: NTAC and component context

The component context consists of *input automata* and *output automata*. Input automata are timed automata that send a message that is received by the NTAC while output automata receive messages from the NTAC. The input and output automata abstract from the timed automata of the other components in the system. This allows for generating TFPGs from an NTAC. Consequently, the reachable behavior, which is used for TFPG construction, is much smaller than the reachable behavior of the system, in particular in systems with concurrency. This, in turn, allows for generating TFPGs from component types with larger behavior models in contrast to the case where the whole system behavior needs to be taken into account.

For the computation of the normal behavior, we use a so-called *initial context*.

Definition 4.2.1 (Initial Context)
Let $\mathcal{A} = \{A_1, ..., A_n\}$ *be an NTAC with* $A_i = (L_i, l_{0i}, C_i, V_i, \Sigma_i, R_i, E_i, I_i)$,

$$V = \bigcup_{i=2}^{n} V_i, \quad \Sigma = \bigcup_{i=2}^{n} \Sigma_i, \quad \textit{and} \quad E = \bigcup_{i=2}^{n} E_i.$$

Let $\Omega_? = \{(a?, d_?) \mid a? \in \Sigma$ *and* $\exists e \in E_{\mathcal{A}}$, $e = (l_x, \varphi, a?, r, \varnothing, d_?, l_y)$, $d_? =$"$v := in$", $v \in V \vee d_? = \epsilon\}$ *and* $\Omega_! = \{(a!, d_!) \mid a! \in \Sigma$ *and* $\exists e \in E_{\mathcal{A}}$, $e = (l_x, \varphi, a!, r, \varnothing, d_!, l_y)$, $d_! =$"$out := v$", $v \in V \vee d_! = \epsilon\}$.

We define the initial context $X_i(\mathcal{A}) = (I_i, O_i)$ *with* $I_i = \{A_{i1}, ..., A_{i|\Omega_?|}\}$ *as the set of initial input automata over* $\Omega_?$ *with* $A_{ij} = (L_{ij}, l_{0ij}, C_{ij}, V_{ij}, \Sigma_{ij}, R_{ij}, E_{ij}, I_{ij})$ *where*

- $L_{ij} = \{l_{ij}\}$,
- $l_{0ij} = l_{ij}$,
- $C_{ij} = \varnothing$,
- $V_{ij} = V$,
- $\Sigma_{ij} = \{a!\}$,
- $R_{ij} = \epsilon$,
- $E_{ij} = \{(l_{ij}, \varnothing, a!, \epsilon, \varnothing, d_{ij}, l_{ij})\}$,

$$d_{ij} = \begin{cases} \text{“}out := v'' & d_? = \text{“}v := in'', v \in V \\ \epsilon & d_? = \epsilon \end{cases}$$

  , $(a?, d_?) \in \Omega_?$, *and*

- $I_{ij} = \varnothing$

$O_i = \{A_{o1}, ..., A_{o|\Omega_!|}\}$ *is the set of output automata over* $\Omega_!$ *with* $A_{oj} = (L_{oj}, l_{0oj}, C_{oj}, V_{oj}, \Sigma_{oj}, R_{oj}, E_{oj}, I_{oj})$ *where*

- $L_{oj} = \{l_{oj}\}$,
- $l_{0oj} = l_{oj}$,
- $C_{oj} = \varnothing$,
- $V_{oj} = V$,
- $\Sigma_{oj} = \{a?\}$,
- $R_{oj} = \epsilon$,
- $E_{oj} = \{(l_{oj}, \varnothing, a?, \epsilon, \varnothing, d, l_{oj})\}$,

$$d_{oj} = \begin{cases} \text{“}v := in'' & d_! = \text{“}out := v'', v \in V \\ \epsilon & d_! = \epsilon \end{cases}$$

  , $(a!, d_!) \in \Omega_!$, *and*

- $I_{oj} = \varnothing$

The initial context consists of one input automaton for each pair of message and variable assignment that is received by the NTAC and one output automaton for each pair of message and variable assignment that is sent by the NTAC. The input and output automata of the initial context do not have any time

constraints, because we cannot compute the absolute firing times of transitions in the NTAC without computing the normal behavior of the NTAC. However, time constraints are not needed to compute the normal behavior, because the times when transitions in the NTAC fire are specified by the time constraints of the NTAC. The omitted time constraints do not affect the computation of the normal behavior, because the timed automata of the initial context can only fire synchronously with the transitions in the NTAC.

The initial context for the timed automaton of Figure 4.2 is shown in Figure 4.5. Figures 4.5(a), 4.5(b), and 4.5(c) show the timed automata that send p4.mode, p1.speed1, and p2.speed2; the messages which are received as inputs by the timed automaton of Figure 4.2. Figure 4.5(d) shows the timed automaton that receives the message p3.posData, which is sent as output message by the timed automaton of Figure 4.2.

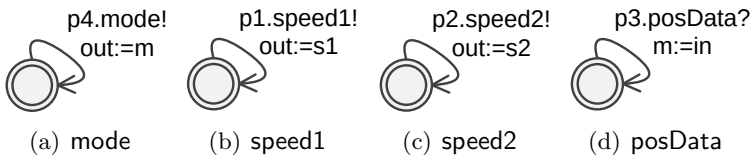


Figure 4.5: Initial context of the component type PosCalc

Note, that we do not test variable assignments. This is done by the reachability analysis that constructs the reachable behavior.

Figure 4.6 shows the zone graph that specifies the normal behavior of the NTAC of Figure 4.2. It contains two paths that correspond to the executions of the NTAC. For later use, we denote the upper path by Path 1 and the lower path by Path 2. Path 1 represents the behavior of the slow drive mode and Path 2 the behavior of the fast drive mode.

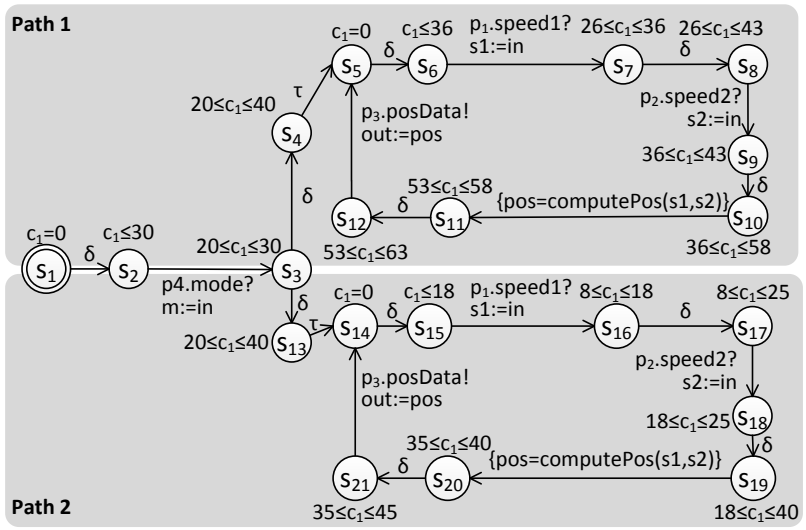


Figure 4.6: Reachable behavior of the NTAC of Figure 4.2

Computing the Failure Behavior

Computing the failure behavior is based on a set of failure contexts. Each failure context injects a unique combination of incoming failures into the NTAC. Incoming failures are modeled by input automata such that they either send no message (service failure) or send the message too early or too late (timing failure).

The basis for the failure contexts is the *refined context*. In contrast to the initial context, the refined context contains one input automaton for each transition of the NTAC that sends a message. Each input automaton has a time constraint that specifies the time when the receiving transition in the NTAC is activated and can thus synchronize with the input automaton. These time constraints are needed to compute time constraints for input automata that inject timing failures into the NTAC. The time constraints of input automata of the refined context are constructed from the clock zones of the normal behavior.

The computation of runtimes in zone graphs requires the time guards of the transitions in the timed automata that correspond to a zone graph transition. Therefore, we first define a function that returns these guards from zone graph transitions.

Definition 4.2.2 (φ_t, λ_t)
Let $Z = (S, s_0, \Sigma', T, V, \mu, \nu)$ be the zone graph (cf. Definition 3.3.5) representing the reachable behavior of the NTAC $\mathcal{A} = \{A_1, ..., A_n\}, n \in \mathbb{N}, A_i = (L_i, l_{0i}, C_i, V_i, \Sigma, R_i, E_i, I_i)$, $i \in \{1, ..., n\}$,

$$L = \bigcup_{i=1}^{n} L_i, \quad \textit{and} \quad E = \bigcup_{i=1}^{n} E_i.$$

Let $f_s : E \to L$ and $f_t : E \to L$ be the source and target functions of edges in the timed automata.

Let $\varphi_T : T \to \mathcal{B}(\mathcal{C})$ be the mapping from transitions in Z to the time guards of the corresponding transitions in $\mathcal{A}$ *and $\lambda_T : T \to C$ the mapping from transitions in Z to the clock resets of the corresponding transitions in $\mathcal{A}$.*

- *Consider $t \in T$ with $t = ((l_a, k_a, \hbar_a), (l_b, k_b, \hbar_b))$.*
- *$E_{l_a,l_b} = \{e \in E \mid f_s(e) = l_{ai}, f_t(e) = l_{bi}, i = 1, ..., n, l_{ai} \neq l_{bi}\}$.*
- *Without loss of generality, let $E_{l_a,l_b} = \{e_1, ..., e_m\} =$ $\{(l_1, \varphi_1, \sigma_1, r_1, \lambda_1, D_{u1}, l'_1), ..., (l_m, \varphi_m, \sigma_m, r_m, \lambda_m, D_{um}, l'_m)\}$, then*

$$\varphi_T(t) = \bigwedge_{i=1}^{m} \varphi_i, \quad \textit{and} \quad \lambda_T(t) = \bigcup_{i=1}^{m} \lambda_i.$$

Definition 4.2.3 (Runtime of a Zone Graph Path)
Let $Z = (S, s_0, \Sigma', T, V, \mu, \nu)$ be the zone graph (cf. Definition 3.3.5) representing the reachable behavior of the NTAC $\mathcal{A} = \{A_1, ..., A_n\}, n \in \mathbb{N}, A_i = (L_i, l_{0i}, C_i, V_i, \Sigma, R_i, E_i, I_i)$, $i \in \{1, ..., n\}$, and

$$E = \bigcup_{i=1}^{n} E_i.$$

Consider a path p in Z with $p = t_1, \ldots, t_l$, $t_i = (s_i, \sigma_i, s_{i+1})$, $s_i = (l_i, k_i, \hbar_i)$.

For the set of clocks $C_p = \{c \in C \mid \exists i \in \{1, ..., l+1\} : c \in \rho(\hbar_i)\}$ of the path p, we partition p into m partitions $t_{11}, \ldots, t_{1n_1}, t_{21}, \ldots, t_{2n_2}, \ldots, t_{m1}, \ldots, t_{mn_m}$ with $p_i = t_{i1}, \ldots, t_{in_i}$ where $\lambda_T(t_{in_i}) \neq \varnothing$ and $\forall j = 1, ..., n_i - 1 : \lambda_T(t_{ij}) = \varnothing$.

For a partition $p_i = t_{i1}, \ldots, t_{in_i}$, its clock zones $\hbar_{i1}, \ldots, \hbar_{i(n_i+1)}$, and for each clock $c_j \in C_p$, we define the following values:

$$\begin{aligned}
t^0_{i,j} &= \lim_{c_j \to \infty} (\exists c_1, \ldots, c_{j-1}, c_{j+1}, \ldots, c_{|C|} \in C_p : \hbar_{i1}) \\
t^1_{i,j} &= \lim_{c_j \to 0} (\exists c_1, \ldots, c_{j-1}, c_{j+1}, \ldots, c_{|C|} \in C_p : \hbar_{in_i} \wedge \varphi_T(t_{in_i}))) \\
t^2_{i,j} &= \lim_{c_j \to 0} (\exists c_1, \ldots, c_{j-1}, c_{j+1}, \ldots, c_{|C|} \in C_p : \hbar_{i1}) \\
t^3_{i,j} &= \lim_{c_j \to \infty} (\exists c_1, \ldots, c_{j-1}, c_{j+1}, \ldots, c_{|C|} \in C_p : \hbar_{in_i} \wedge \varphi_T(t_{in_i}))
\end{aligned}$$

The minimum and maximum propagation times of path p are defined by $\vartheta_{\min}(p)$ and $\vartheta_{\max}(p)$:

$$\begin{aligned}
\vartheta_{\min}(p) &= \max_{j \in \{1,\ldots,|C_p|\}} \sum_{i=1}^{m} (t^1_{i,j} - t^0_{i,j}) \\
\vartheta_{\max}(p) &= \min_{j \in \{1,\ldots,|C_p|\}} \sum_{i=1}^{m} (t^3_{i,j} - t^2_{i,j})
\end{aligned}$$

The propagation time interval *is defined by*

$$\Delta\vartheta(p) = [\vartheta_{\min}(p), \vartheta_{\max}(p)].$$

Since the clocks of the timed automata of the NTA may be reset to zero by clock resets, the values of the clocks along the path of the zone graph do not reflect the amount of time that was spent on a path directly. Instead, they only measure the time that elapsed after the last reset. Thus in order to compute runtimes of paths in zone graphs, we need to sum up the time that elapsed between two

resets along a path of the zone graph. This is done by partitioning the paths into sub paths that start and end at transitions where a reset occurred.

The runtime of the partitions are computed from the differences between the clock zones of the last and the first state of each transition. However, the clock zone of the last state cannot be found in the zone graph: Each transition in a timed automaton is represented by two transitions in the zone graph. The first transition in the zone graph modifies the clock zones according to the state invariants of the target state of the transition in the timed automaton. The second transition modifies the clock zone according to the time guards of the timed automaton transition. If the transition in the timed automaton resets a clock c, the clock zone of the target state of the second transition in the zone graph is always $c = 0$. We therefore compute the clock zone of the last state of a partition by the conjugation of the clock zone of the penultimate state and the time guards of the transitions in the NTA that correspond to the last transition in the partition.

The maximum propagation time ϑ_{max} is computed as follows: For each clock c_j, we compute the minimum value $t^2_{i,j}$ that satisfies the clock zone $\hbar_{i1}$ of the first state of the partition p_i and the maximum value $t^3_{i,j}$ that satisfies the clock zone $\hbar_{in_i}$ of the penultimate state of p_i. This clock value must also be a valid clock value for all other clock zones of the state. Then, we compute the runtime of the path as the sum of the differences of $t^3_{i,j} - t^2_{i,j}$ of all partitions for each clock. We take the minimum over all sums of all clocks, because we must choose the clock values such that all clock zones are satisfied. This corresponds to the minimum runtime[1] over all clocks. The clock zone of the last state is computed from the clock zone of the penultimate state and the timed guards of the last transition of the partition as described above.

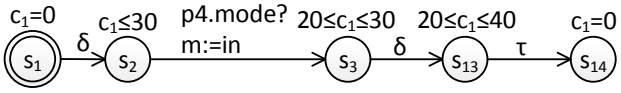
For the minimum propagation time ϑ_{min}, we compute for each clock the maximum value $t^0_{i,j}$ that satisfies the clock zone $\hbar_{i1}$ of the first state of the partition p_i and the minimum value $t^1_{i,j}$ that satisfies the clock zone $\hbar_{in_i}$ of the penultimate state of p_i. This clock value must also be a valid clock value for all other clock zones of the state. Then, we compute the runtime of the path as the sum of the differences of $t^1_{i,j} - t^0_{i,j}$ of all partitions for each clock. Of all sums of all clocks, we take the maximum, because we must choose the clock values such that all clock zones are satisfied. This corresponds to the maximum runtime[2] over all clocks. The clock zone of the last state is computed from the clock zone of the penultimate state and the timed guards of the last transition of the partition as described above.

The naive approach would be to measure the path runtimes with a global clock. This, however, does not work with clock zones. In this case, the global clock would cause infinite paths in the reachable state space.

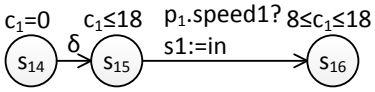
[1] We take the greatest value of the minimum values and the smallest value of the maximum values to satisfy all clock zones.

[2] We take the smallest value of the minimum values and the greatest value of the maximum values to satisfy all clock zones.

Figure 4.7 shows the two partitions which are created when computing the minimum and maximum time needed to traverse Path 2 from the initial state to transition (s_{15}, s_{16}) where p1.speed1 is received. These times are needed to create the clock constraints for the input automaton that sends p1.speed1.



(a) Partition 1



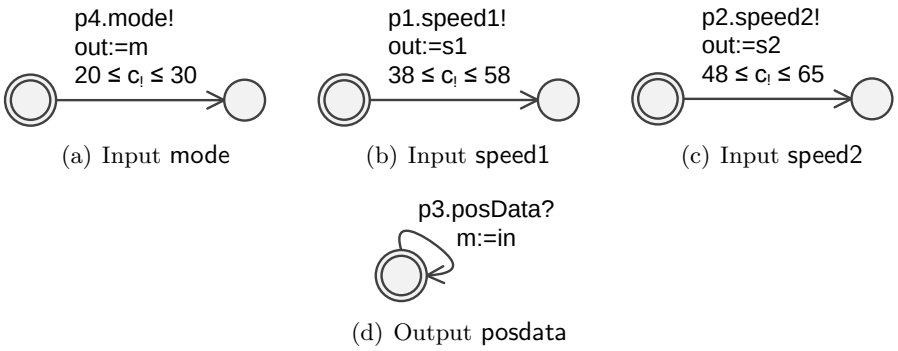
(b) Partition 2

Figure 4.7: Partitions of the path from s1 to s16

For Partition 1 of Figure 4.7(a) we compute the traversal time as follows. The clock zone of the first state in Partition 1 is $c_1 = 0$. The clock zone of the penultimate state of Partition 1 is $20 \leq c_1 \leq 40$. To compute the clock zone of s_14, we need the time guards of the corresponding transition of the NTAC. The only corresponding transition of (s_{13}, s_{14}) in the zone graph of Figure 4.6 is (l_2, l_3) in the component automaton of Figure 4.2. This transition has the time guard $c_1 \geq 30$. We compute the clock zone of the last state in Partition 1 by $(20 \leq c_1 \leq 40) \wedge (c_1 \geq 30) = 30 \leq c_1 \leq 40$. The propagation time interval for Partition 1 is therefore $[30 - 0, 40 - 0] = [30, 40]$. The propagation time interval $[8, 18]$ for Partition 2 is computed analogously. Thus the minimum firing time of transition (l_3, l_4) in the component automaton of Figure 4.2 is $30 + 8 = 38$ and the maximum firing time is $40 + 18 = 58$. These times are used to model the refined input automaton for transition (s_{15}, s_{16}).

Figure 4.8 shows the refined context that was constructed from Path 2 of the normal behavior of the component type PosCalc of Figure 4.6. Figures 4.8(a), 4.8(b), and 4.8(c) show the input automata of the refined context of Path 2. Each input automaton consists of two locations and one transition that sends a message. The transition has a time guard. In Figure 4.8(a), the transition has the time guard $20 \leq c_! \leq 30$. This is the absolute time at which p4.mode is received by the NTAC. This time interval is specified by the clock zone $20 \leq c_1 \leq 30$ at state s_3 in the normal behavior of Figure 4.2.

The output automata of the refined context have one location and a self-transition that receives a message. The activation of the transitions is not limited by time constraints, because output automata are not used to model timing failures. Figure 4.8(d) shows the output automaton of message p3.posData of the refined context.



(a) Input mode (b) Input speed1 (c) Input speed2

(d) Output posdata

Figure 4.8: Refined context of Path 2 of Figure 4.6

Definition 4.2.4 (Refined Context)
Let $Z = (S, s_0, \Sigma, T, V, \mu, \nu)$ *be the zone graph that represents the normal behavior of an NTAC* $\mathcal{A} = \{A_1, ..., A_n\}, n \in \mathbb{N}, A_i = (L_i, l_{0i}, C_i, V_i, \Sigma, R_i, E_i, I_i), i \in \{1, ..., n\}$,

$$V_{\mathcal{A}} = \bigcup_{i=1}^{n} V_i, \quad R_{\mathcal{A}} = \bigcup_{i=1}^{n} R_i, \quad \textit{and} \quad E_{\mathcal{A}} = \bigcup_{i=1}^{n} E_i,$$

and $X_i(\mathcal{A}) = (I_i, O_i)$ *the initial context of* $\mathcal{A}$.

Without loss of generality let $t = (s_k, a?, s_l), t \in T, a? \in \Sigma$ *be a transition in* Z *that corresponds to the edge* $e = (l_x, \varphi, a?, r, \lambda, d, l_y)$, $e \in E_{\mathcal{A}}$.

We define the input automaton
$A_! = (L_!, l_{0!}, C_!, V_!, \Sigma_!, R_!, E_!, I_!)$ *of* t *with*

- $L_! = \{l_{0!}, l_{1!}\}$,
- $C_! = \{c_!\}$,
- $V_! = V_{\mathcal{A}}$,
- $\Sigma_! = \{a!\}$,
- $R_! = \epsilon$,
- $E_! = \{(l_{0!}, \varphi_!, a!, \epsilon, \varnothing, d_!, l_{1!})\}$, *with* $\varphi_! = c_! \geq \vartheta_{min}(p) \wedge c_! \leq \vartheta_{max}(p)$, *for all paths* $p = t_j, ..., t$ *with* $t_j = (s_0, \sigma_j, s_m)$, $s_m \in S$,

$$d_! = \begin{cases} \text{“}out := v'' & d = \text{“}v := in'' \\ \epsilon & d = \epsilon \end{cases}$$

  and

- $I_t(l_{0!}) = I(l_{1!}) = \varnothing$.

Let $T_? = \{t_1, ..., t_l\}$ *with* $t_i = (s_j, a?, s_k), s_j, s_k \in S, i \in \{1, ..., l\}, l \in \mathbb{N}, a? \in \Sigma$. *Let* $E_? \subseteq E_\mathcal{A}$ *be the edges that correspond to the transitions in* $T_?$.

We define the refined context *by* $X(\mathcal{A}) = (\mathcal{I}, \mathcal{O})$ *with* $\mathcal{I} = \{A_{i1}, ..., A_{i|T_?|}\}$ *where* A_{ij} *is the input automaton of* $t_j \in T_?$, *and the set of* output automata $\mathcal{O} = O_i$.

Our definition of input automata allows input messages to be sent only once. Consequently, only the first run of a cycle in the reachable behavior is taken into account. We leave the analysis of multiple runs in cycles of the normal behavior for future work.

The refined context is designed such that it adds as little behavior as possible to the NTAC. All input automata share a common clock which starts with zero at the same time that the NTAC starts. By using this common clock instead of a unique clock for each input automaton, we reduce the number of clock zones of the reachable behavior. Additionally, we construct one output automaton for each pair of variable assignment and message that is sent by the NTAC. The output automata have neither clocks nor time constraints. The shared clocks of input automata and the omission of time constraints in output automata keep the failure behaviors small. Consequently, the identification of relations between incoming and outgoing timing and service failures becomes more efficient.

Computing the failure behavior is based on failure contexts. A failure context is constructed by replacing one or more input automata of the refined context by failure automata. Failure automata are constructed by modifying input automata such that they either send no message (service failure) or send their message too early or too late (timing failure).

Definition 4.2.5 (Failure Automata)
Let $\mathcal{A}$ *be an NTAC and* $Z = (S, s_0, \Sigma, T, V, \mu, \nu)$ *the zone graph that represents the normal behavior of* $\mathcal{A}$. *Let* $X(\mathcal{A}) = (\mathcal{I}, \mathcal{O})$ *be the refined context of* $\mathcal{A}$. *Without loss of generality* $A = (L, l_0, C, V, \Sigma, R, E, I), A \in \mathcal{I}$ *is an input automaton of the refined context with*

- $L = \{l_0, l_1\}$,
- $C = \{c\}$,
- $V = \{v\}$,
- $\Sigma = \{a?\}$,
- $R = \{\epsilon\}$,
- $E = \{(l_0, \varphi, a!, \epsilon, \varnothing, d, l_1)\}$, *with* $\varphi = b_1 \wedge b_2$, $b_1 = c \sim_1 n_1$, $b_2 = c \sim_2 n_2$, $\sim_1 \in \{\geq, >\}$, $\sim_2 \in \{\leq, <\}$, $n_1, n_2 \in \mathbb{N}$, *and*
- $I = \varnothing$.

We define $s(A) = (\{l_s\}, l_s, \varnothing, V, \{\tau\}, \{\epsilon\}, \varnothing, \varnothing)$ *the* service failure automaton *of* A*.*

We define $e(A) = (L_e, l_{e0}, C, V, \Sigma, R, E_e, I)$ *the* early timing failure automaton *of* A *where*

- $L_e = \{l_{e0}, l_{e1}\}$ *and*
- $E_e = \{(l_{e0}, \varphi_e, a!, \epsilon, \varnothing, d, l_{e1})\}$, $\varphi_e = b_{e1} \wedge b_{e2}$, $b_{e1} = c \geq 0$, *and* $b_{e2} = c \sim_{e2} n_1$,

$$\sim_{e2} = \begin{cases} \leq & \sim_1 = > \\ < & \sim_1 = \geq \end{cases}$$

We define $l(A) = (L_l, l_{l0}, C, V, \Sigma, R, E_l, I)$ *the* late timing failure automaton *of* A *where*

- $L_l = \{l_{l0}, l_{l1}\}$ *and*
- $E_l = \{(l_{l0}, \varphi_l, a!, \epsilon, \varnothing, d, l_{l1})\}$, $\varphi_l = b_{l1} \wedge b_{l2}$, $b_{l1} = c \sim_{l1} n_2$,

$$\sim_{l1} = \begin{cases} \geq & \sim_2 = < \\ > & \sim_2 = \leq \end{cases}$$

, $b_{l2} = c < \infty$.

A service failure occurs if a message that is expected is never received. We thus model an incoming service failure by a timed automaton without transitions. To model a context in which the message a of the input automaton will never be sent, we replace all input automata that carry this message a by such a timed automaton.

A timing failure occurs if a message is not delivered within an expected time interval. Instead of testing every clock value of each clock separately, we analyze all equivalent clock values of one timing failure by one symbolic execution. For this, we exploit the fact that intervals of equivalent clock values are represented by clock zones in the reachable behavior. This enables testing all possible timing failures by two failure automata: one which models all cases where the message is sent too early (incoming early timing failure) and one which models all cases where the message is sent too late (incoming late timing failure). We thus model an incoming timing failure by altering the time guard of an input automaton such that the time when the output message is sent is not within the expected time interval. For each input automaton, we generate a failure automaton provoking an early timing failure and another one provoking a late timing failure.

To model a late timing failure for the input message `p1.speed1`, we modify the time guard of the input automaton of Figure 4.8(b). The time guard of the input automaton (cf. Figure 4.8(b)) is $38 \leq c_! \leq 58$. The time guard of the failure automaton is $c_! > 58$ and thereby enables the transition only when its clock

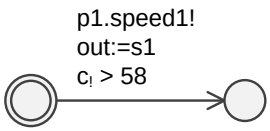


Figure 4.9: Late timing failure for speed1

has a time value which is greater than the time values of the input automaton. Consequently, the input message p1.speed1 will always be sent too late.

With the help of the failure automata, we are now able to construct the failure contexts. The NTAC and each failure context build an NTA. We construct a failure behavior from each of these NTA.

Definition 4.2.6 (Failure Context)
Let $\mathcal{A}$ be an NTAC, $Z = (S, s_0, \Sigma, T, V, \mu, \nu)$ the zone graph representing the reachable behavior of $\mathcal{A}$ and $X(\mathcal{A}) = (\mathcal{I}, \mathcal{O})$ the refined context of $\mathcal{A}$. Let $p = t_1, ..., t$ with $t_1 = (s_0, \sigma, s_k)$, $s_k \in S$, and $t_1, ..., t \in T$ be a path in Z.

Let A be the input automaton of t and $F(A) = \{A_!, s(A), e(A), l(A)\}$ the failure automata of A.

We define the failure contexts *of p by $\mathcal{X}_f(p) = (\mathcal{I}_f, \mathcal{O}_f)$ where $\mathcal{I}_f = F(A_1) \times ... \times F(A_{|T_?|})$ with $T_? = \{t_1, ..., t_l\}$, $t_i = (s_j, a?, s_k), s_j, s_k \in S, i \in \{1, ..., l\}, l \in \mathbb{N}$ and $\mathcal{O}_f = \mathcal{O}$.*

The set of failure automata of each input automaton contains the input automaton itself. This is needed to construct the set of failure contexts from the sets of failure automata. Input automata are part of those failure contexts of which not every input represents and incoming failure.

We actually would need to construct a failure behavior for each combination of incoming failures of the NTAC. The NTAC has $|\mathcal{X}_f(p_1) \times ... \times \mathcal{X}_f(p_n)|$ combinations of incoming failures. However, the combinations of input messages are limited by the paths in the normal behavior. Since the injected failures are derived from input messages, the number of combinations of injected failures are also limited by the paths in the normal behavior. We thus only need to inject incoming failures for each path of the normal behavior separately. The number of combinations of injected failures is thereby reduced to $|\mathcal{X}_f(p_1)| + ... + |\mathcal{X}_f(p_n)|$.

To inject only failures for a path p of the NTAC, the failure context only contains failure and input automata of p. The output automata are the same as in the refined context. They enable the sending of each output message by the NTAC at any time. This, in turn, enables the occurrences of all possible outgoing timing failures.

Using the failure behaviors, we identify outgoing timing and service failures.

Definition 4.2.7 (Failure Classes of a Zone Graph)
Let $\mathcal{A}$ be an NTAC and $Z = (S, s_0, \Sigma, T, V, \mu, \nu)$ the zone graph representing the reachable behavior of $\mathcal{A}$, $p = t_j, ..., t_k, ...t_l, t_i \in T, i \in \{j, ..., l\}$ be a path in Z, and $\hat{Z} = (\hat{S}, s_0, \hat{\Sigma}, \hat{T}, \hat{V}, \hat{\mu}, \hat{\nu})$ a zone graph representing a failure behavior of p.

$\hat{Z}$ contains an outgoing service failure*, iff $\exists t \in T, t = (s_a, \sigma, s_b), s_a, s_b \in S$, $\sigma \in \Sigma \nexists \hat{t}' \in \hat{T}, \hat{t}' = (\hat{s}_a, \hat{\sigma}, \hat{s}_b)$, $\hat{s}_a, \hat{s}_b \in \hat{S}$, $\hat{\sigma} \in \hat{\Sigma} : \sigma = \hat{\sigma}'$.*

Without loss of generality, consider the transition $\hat{t} = (\hat{s}_j, \hat{\sigma}, \hat{s}_k) \in \hat{T}$, $\hat{s}_j, \hat{s}_k \in \hat{S}$, $\hat{\sigma} \in \hat{\Sigma}$.

$\hat{t}$ produces an outgoing early timing failure*, iff $\nexists t \in T$ with $t = (s_l, \hat{\sigma}, s_m)$, $s_l, s_m \in S$, $\hat{\sigma} \in \Sigma$: $\hat{h}_j - h_l = \perp$, and $\exists t' \in T$ with $t' = (s_p, \hat{\sigma}, s_q)$, $s_p, s_q \in S$, $\hat{\sigma} \in \Sigma$: $\hat{h}_j - \hat{h}_p \Uparrow \neq \perp$.*

$\hat{t}$ produces an outgoing early timing failure*, iff $\nexists t \in T$ with $t = (s_l, \hat{\sigma}, s_m)$, $s_l, s_m \in S$, $\hat{\sigma} \in \Sigma$: $\hat{h}_j - h_l = \perp$, and $\exists t' \in T$ with $t' = (s_p, \hat{\sigma}, s_q)$, $s_p, s_q \in S$, $\hat{\sigma} \in \Sigma$: $\hat{h}_j - \hat{h}_p \Downarrow \neq \perp$.*

The operator $\Downarrow$ ($\Uparrow$) returns all values of a clock zone and all values before (after) that clock zone [BY03].

We find an outgoing service failure if the normal behavior outputs a message a that is not output by the failure behavior. An outgoing timing failure is found, if the failure behavior outputs a message a earlier or later than the normal behavior.

If we inject a service failure on port p4 into the NTAC of the component type PosCalc (cf. Figure 4.2), the NTAC will never leave its initial state. Consequently, there exists no path that ends at a transition in the failure behavior and that sends p3.posCalc. This means, an outgoing service failure occurred at port p3. A new TFPG is created with the outgoing service failure on port p3 as top node. The identified relation between this outgoing service failure and the incoming service failure on port p4 is stored in this TFPG. The propagation time interval between both failures is $[73, 103]$.

We ensure storing minimal combinations of incoming failures for each outgoing failure by injecting incoming service and timing failures in the following ordering: We start with one incoming failure per failure context. When all cases of one incoming failure have been analyzed, we add one incoming failure and check all combinations of two incoming failures. Each time we have analyzed all combinations of failures of a certain size, we increase the size by one. Each time we find an outgoing failure for a combination of incoming failures, we save this combination. A new combination is discarded if it results in an outgoing failure, which was already caused by a combination of failures, which is a subset of the new combination. We thus avoid redundancies in the TFPGs: All combinations containing another combination would also lead to the same outgoing failure [Rau01]. By discarding these combinations, our approach keeps

the TFPG as small as possible. This makes the analysis of failure propagation more efficient.

4.2.2 Value Failures

Identifying relations between incoming and outgoing value failures is based on the slicing of extended finite state machines (EFSM) of Androutsopoulos et al. [ACH+12]. It is the only approach for slicing nonterminating automata, which makes it the only suitable approach for embedded real-time systems. The input is an EFSM and a slicing criterion. The slicing criterion specifies a state in the EFSM and a variable whose dependencies are analyzed. The resulting slice is also an EFSM, but it only contains the parts, which affect the variable of the slicing criterion.

We map the NTAC to an EFSM to apply slicing. We then identify relations between incoming and outgoing value failures using the slice. We map the slice back to the NTAC to compute the propagation time intervals between the identified incoming and outgoing value failures. For better readability, we use the *component automaton* instead of the NTAC. The component automaton is the product automaton of the NTAC.

We first repeat the definition of EFSMs of [ACH+12] before defining the mapping in Def. 4.2.9.

Definition 4.2.8 (Extended Finite State Machine [ACH+12])
An Extended Finite State Machine (EFSM) M is a tuple (S, s_0, T, E, V', v_0) where: S is a set of states; $s_0 \in S$ is the initial state; T is a set of transitions; E is a set of events, where each event is an atomic message or signal, possibly parameterized; V' is a store represented by a set of variables; and v_0 is a mapping from the variable names to the initial value of these variables. Transitions have a source state $\text{source}(t) \in S$*, a target state* $\text{target}(t) \in S$ *and a label* $\text{label}(t)$*. Transition labels are of the form $e[g]/a$ where $e \in E$; g is a guard (we assume a standard condition language); and a is a sequence of messages (we assume a standard expression language including assignments). All parts of a label are optional.*

Definition 4.2.9 (Mapping Timed Automata to EFSMs)
Given are the timed automaton $A = (L, l_0, C, V, \Sigma, R, E, I)$ and the EFSM $M = (S, s_0, T, E', V', v_0)$. Let $f_s : E \to L$ and $f_t : E \to L$ be the source and target functions of edges in the timed automaton.

We define the graph morphism $\mu = (\mu_L, \mu_V, \mu_E, \mu_\Sigma)$ where

- $\mu_L : L \to S$ *(bijective),* $\mu_L(l_0) = s_0$ *(bijective),*
- $\mu_V : V \to V'$ *(bijective),*

- $\mu_E : E \to T$ *(bijective), with* $\text{source}(\mu_E(e)) = \mu_L(f_s(e))$, $\text{target}(\mu_E(e)) = \mu_L(f_t(e))$, *and* $e \in E$, *and*
- $\mu_\Sigma : \Sigma \cup D_u \to E'$, *and*

$v_0(\text{v}) = 0 \ \forall \text{v} \in \text{V}'$.

μ maps a timed automaton to an isomorphic EFSM by omitting clocks and clock constraints.

We compute the slice for each variable v that is sent by the component automaton. The variable assignments that are part of the slice are those which influence v.

Definition 4.2.10 (Computing the Slice)
The slice M' of an EFSM M is computed by the function slice(M, C) where $C = (t, V)$ is a slicing criterion with the transition t and the set of variables V.

The slice represents the part of the EFSM that influences the values of the variables in set V after transition t was executed.

In order to compute propagation times between incoming and outgoing value failures, we need to identify the transitions of the component automaton that correspond to the transitions in the slice. However, the slicing algorithm does not guarantee that the slice is an isomorphic subgraph of the component automaton, because states may be merged and transitions my by deleted during slicing. However, the states of the original EFSM are stored in the merged states of the slice. We thus identify the part of the component automaton that corresponds to the slice by mapping the states of the slice back to the locations of the component automaton.

Definition 4.2.11 (Construction of timed automata from EFSMs)
Given are the timed automaton $A = (L, l_0, C, V, \Sigma, R, E, I)$, the EFSM $\mu(A) = M$, and a slicing criterion C.

Without loss of generality let $M_s = slice(M, C)$ with $M_s = (S_s, s_{0s}, T_s, E'_s, V'_s, v_{0s})$ be a slice of M.

We define $\nu : M_s \mapsto A_s$ the mapping of the slice to a subgraph A_s of A with $A_s = (L_s, l_{0s}, C, V, \Sigma, R, E_s, I)$ where

- $L_s = \mu_L^{-1}(S_s)$
- $l_{0s} = \mu_L^{-1} s_{0s}$
- $E_s = \{e \in E \mid \mu_L^{-1}(f_s(e)) \in L_s$, *and* $\{e \in E \mid \mu_L^{-1}(f_t(e)) \in L_s\}$

We construct the subgraph A_s which corresponds to the slice M of the timed automaton A by the reverse application of the mapping μ_L (cf. Definition 4.2.9). μ_l^{-1} maps EFSM states to locations of the timed automaton. Further, all edges that connect the nodes of A_s in A are transfered to A_s.

With these functions, we can now construct TFPGs with incoming and outgoing value failures from the component automaton. The computations are formalized in Algorithms 1 - 3. Algorithm 1 maps the component automaton to the EFSM and triggers the construction of TFPGs (implemented by Algorithm 2) for each outgoing variable of the EFSM. Algorithm 2 constructs the structure of the TFPG and triggers Algorithm 3, which computes the propagation time intervals.

Next, a self-transition is added to the start state of the component automaton (Line 2). This self-transition is used as the starting transition of the slicing criterion (cf. Definition 4.2.10). By using this transition in the slicing criterion, the slicing will always start at the initial state of the component automaton and consequently take all input variables into account. Then, the EFSM of the component automaton is constructed as defined in Definition 4.2.9 (Line 3). Afterwards, a TFPG is constructed for each port p_{out} of the component at which messages are sent (Lines 4 - 7). For this Algorithm 2 is used. The result of Algorithm 1 is a set of TFPGs for component type k that contains a TFPG for each port at which variables are output.

Algorithm 1 ConstructTFPGWithValueFailures

Require: $A = (L, l_0, C, V, \Sigma, R, E, I)$ the component automaton,
k the component type,
$\mathcal{V}(sys) = (\text{sys}, \mathcal{E}, \mathcal{F}, f_{\mathcal{E}}, f_{\mathcal{F}})$ the error and failure specification
Ensure: $\mathcal{G}$ the generated TFPGs

1: $\mathcal{G} = \varnothing$
2: create edge $e_0 = (l_0, \varnothing, \tau, \epsilon, \varnothing, l_0)$, $E = E \cup e_0$
3: $M = \mu(A)$ with $M = (S, s_0, T, E', V', v_0)$
4: **for all** p_{out} with $\exists a \in \Sigma$, $\exists d_! : a! = p_{out}.m!$, $d_!$ ="$out := v_{out}$", $v_{out} \in V'$ **do**
5: $G = ConstructTFPGStructure(p_{out}, k, A, M, \mathcal{V}(sys))$
6: $\mathcal{G} = \mathcal{G} \cup G$
7: **end for**
8: **return** $\mathcal{G}$

Algorithm 2 constructs the TFPG of a port p_{out} of the component type k. The algorithm first creates an empty TFPG-tuple (Line 1). This tuple is filled during the execution of the algorithm. Next, the top part of the TFPG is created (Lines 2 - 5). The top part comprises the node with the outgoing failure, an OR-node and an edge which connects both nodes. The propagation time interval of this edge is set to $[0, 0]$. Afterwards, the algorithm computes for each outgoing variable of the port p_{out} which incoming value failures causes a value failure on this outgoing variable (Lines 6-15). Therefore, the algorithm

first computes the slice of each outgoing variable (Line 7). For each incoming variable that is part of this slice, an incoming value failure and its node are created and connected to the OR-node of the top part of the TFPG (Lines 9 - 14). In Line 13, Algorithm 3 is used to compute the propagation time interval for the edge which connects the incoming value failure with the OR-node in the TFPG.

Algorithm 2 ConstructTFPGStructure

Require: p_{out} the port,
k the component type,
$A = (L, l_0, C, V, \Sigma, R, E, I)$ the component automaton,
$M = (S, s_0, T, E', V', v_0)$ the EFSM,
$\mathcal{V}(sys) = (\text{sys}, \mathcal{E}, \mathcal{F}, f_{\mathcal{E}}, f_{\mathcal{F}})$ the error and failure specification
Ensure: G the generated TFPG

1: create empty TFPG $G = (V_G, E_G, f_s, f_t, I_G, l_G, \iota, \eta)$ with $V_G = \varnothing$, $E_G = \varnothing$, $I_G = \varnothing$
2: create outgoing failure $f_o = f^o_{k.p_{out},v}$, $\mathcal{F} = \mathcal{F} \cup f_o$
3: create failure node v_{Go} with $l(v_{Go}) = f_o$, $V_G = V_G \cup v_{Go}$
4: create operator node v_{Gor} with $l(v_{Gor}) = OR$, $V_G = V_G \cup v_{Gor}$
5: create TFPG-edge $e_{Gor} = (v_{Gor}, v_{Go})$, $\iota(e_{Gor}) = [0, 0]$, $E_G = E_G \cup e_{Gor}$
6: **for all** v_{out} with $\exists t \in T$:"$out := v_{out}$"$\in label(t)$ and $p_{out}.m! \in label(t)$ **do**
7: $M_s = slice(M, [t_0, v_{out}])$ with $t_0 = \mu_E(e_0)$ and $M_s = (S_s, s_{s0}, T_s, E'_s, V'_s, v_{s0})$
8: $F_i = \varnothing$
9: **for all** $v_{in} = \{v'_s \in V'_s \mid \exists t_s \in T_s,\ label(t_s) = e[g]/a$, and $e = p_{in}.m?$ "$v_{in} := in$"$\}$ **do**
10: create incoming value failure $f = f^i_{k.p_{in},v}$, $F_i = F_i \cup f$, $\mathcal{F} = \mathcal{F} \cup f$
11: create failure node v_{Gi} with $l(v_{Gi}) = f$, $V_G = V_G \cup v_{Gi}$
12: create TFPG-edge $e_{Gi} = (v_{Gi}, v_{Gor})$, $E_G = E_G \cup e_{Gi}$
13: $G = AddPropagationTimeInterval(v_{in}, v_{out}, M_s, A, G)$
14: **end for**
15: **end for**
16: **return** G

Algorithm 3 first identifies the part of the component automaton that corresponds to the slice using the mapping ν defined in Definition 4.2.11 (Line 1). Then, the propagation time intervals for all paths between the incoming variable v_{in} and the outgoing variable v_{out} is computed (Lines 3 - 6). For the interval, which is added to the TFPG, the minimum propagation time is the minimum of the minimum propagation of all paths and the maximum propagation time the maximum of the maximum propagation times of all paths (Line 5). In Line 7 the propagation time interval is added to the TFPG-edge.

Note, that we do not evaluate the variable assignments of transitions in the component automaton. We rather assume that each incoming variable affects the returned value of a variable assignment, because we do not know, which combinations of incoming variables in a variable assignment affect the outgoing

Algorithm 3 AddPropagationTimeIntervals

Require: v_{in} the incoming variable,
v_{out} the outgoing variable,
$M_s = (S_s, s_{s0}, T_s, E'_s, V'_s, v_{s0})$ the slice
$A = (L, l_0, C, V, \Sigma, R, E, I)$ the component automaton
$G = (V_G, E_G, f_s, f_t, I_G, l_G, \iota, \eta)$ the TFPG
Ensure: G the TFPG extended by a propagation time interval

1: $A_s = \nu(M_s)$ with $A_s = (L_s, l_{0s}, C, V, \Sigma, R, E_s, I)$
2: create initial propagation time interval $\delta = [0, 0]$
3: **for all** paths $p = e_h, ..., e_j$ with $e_h = (l_h, \varphi_h, a_h?, r_h, \lambda_h, d_h, l'_h)$ with $d_h =$"$v_{in} := in$" and $e_j = (l_j, \varphi_h, a!, r_h, \lambda_h, d_j, l'_j)$ with $d_j =$ "$out := v_{out}$" **do**
4: $\delta_{tmp} = \Delta\vartheta(p)$ with $\delta_{tmp} = [t_{min,tmp}, t_{max,tmp}]$
5: $\delta = [min(t_{min}, t_{min,tmp}), max(t_{max}, t_{max,tmp})]$
6: **end for**
7: $\iota(e) = \delta$, with $l(e) = f^i_{k.p_{in},v}$
8: **return** G

variable. We consequently combine all incoming value failures by a logical OR in the TFPG. This is a pessimistic but safe assumption.

Consider for example Path 1 of the normal behavior of PosCalc. We compute the slice of Path 1 with the slicing criterion $[e_0, pos]$ where e_0 is the self-transition added to the initial location of the component automaton (cf. Figure 4.10).

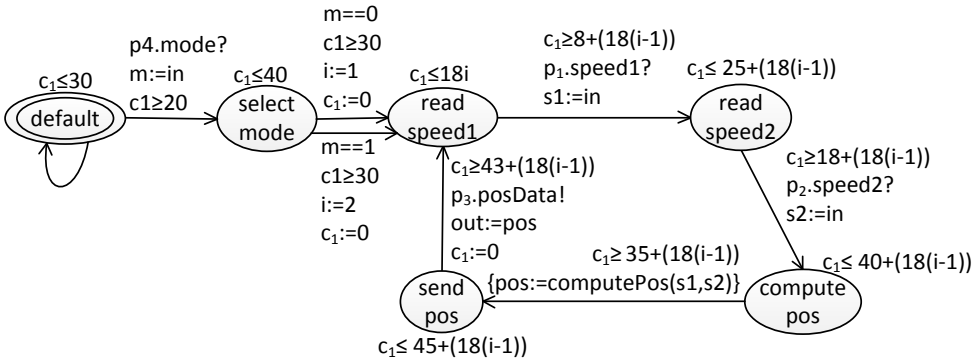


Figure 4.10: Component automaton specifying the behavior of the component type PosCalc extended by a self-transition at the initial state

Figure 4.11 shows the slice of the component automaton of Figure 4.10 and the slicing criterion $[e0, pos]$. We extract the variables that affect the variable pos from the variable assignments at the transitions of the slice. In our example, these are s1:=in and s2:=in. Consequently, the value of pos is affected by the values of the incoming variables s1 and s2. We conclude that a value failure on variable pos is caused by a value failure on s1 or s2.

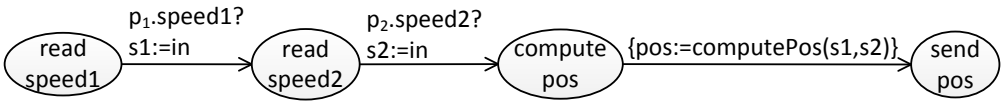


Figure 4.11: Slice of the component automaton of Figure 4.2 for the slicing criterion $[e_0, pos]$

To construct the TFPG, we extract the port names from the messages at the transitions of the slice. For example, s1 is transmitted via port p1, because it is received with the synchronization p1.speed1?. A value failure on port p3 where pos is output is caused by a value failure on port p1 or p2 where s1 and s2 are input.

We construct the subgraph of the component automaton that corresponds to the slice of Figure 4.11 to compute the propagation times. Figure 4.12 shows the subgraph of the component automaton that corresponds to the slice of Figure 4.11. This subgraph is constructed as defined in Definition 4.2.11. This subgraph does not contain the transition at which the variable pos is output. However, this transition is required for the computation of the propagation time interval, because this is the transition where the outgoing value failure occurs. We therefore extend the subgraph of Figure 4.12 by all paths that lead to the output of the variable pos. The result is shown in Figure 4.13.

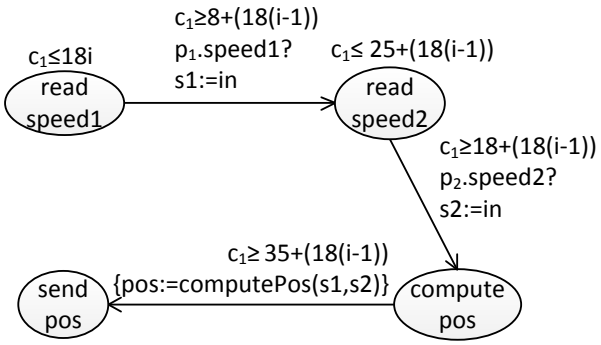


Figure 4.12: Subgraph of the component automaton of Figure 4.2 that corresponds to the slice of Figure 4.11

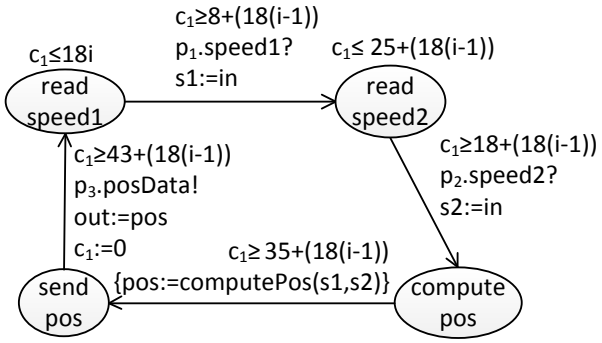


Figure 4.13: Subgraph of the component automaton of Figure 4.2 that corresponds to the slice of Figure 4.11 extended by the path to the output of variable pos

The TFPG of Figure 4.14 shows the TFPG which results from the evaluation of the slice of Figure 4.11. This TFPG specifies how long value failures need to propagate through a component instance of the type PosCalc.

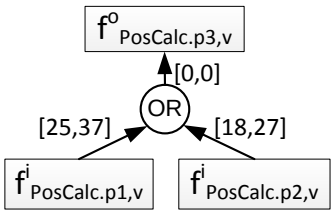


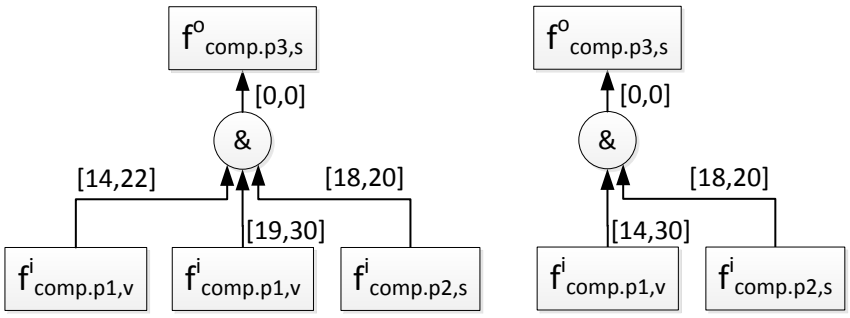
Figure 4.14: TFPG of the outgoing value failure of the component type PosCalc

4.3 Post-processing the Generated TFPGs

A TFPG of a failure class may contain the same combination of incoming errors more than once. This redundant information must be removed to allow for a correct computation of hazard occurrence probabilities and to minimize the size of the TFPG.

There may be two incoming failure variables of the same class at the same port but with different propagation time intervals that are connected by an AND-node. These two failure variables and their propagation time intervals are united.

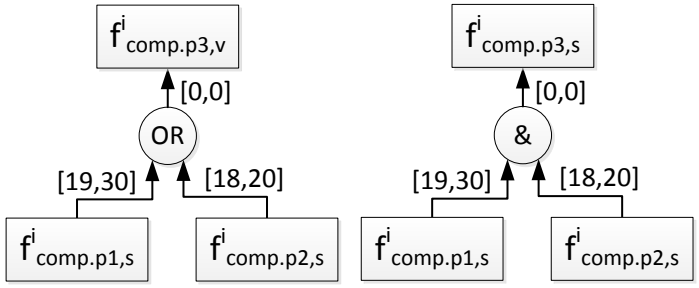
Figure 4.15(a) shows an example where the incoming value failure variable $f^i_{comp.p1,v}$ occurs twice. Such a TFPG is constructed, if there are several input messages at the same port that are received at different points in time. The problem is solved by uniting both nodes that are labeled with this failure variable and uniting the propagation time intervals. In case the intervals, which are united, do not overlap, the new interval is built from the minimum and maximum values of the two intervals.



(a) Before (b) After

Figure 4.15: Incoming failures with overlapping propagation time intervals

It may also be the case that different TFPGs contain the same combination of incoming failure variables where the incoming failure variables are connected by AND in one TFPG and by OR in the other. An example of this case is shown in Figure 4.16. The TFPGs both have the same incoming failures service failure p_1 and service failure p_2 but different outgoing failures. In Figure 4.16(a), they are connected by OR and cause an outgoing value failure. In Figure 4.16(b), they are connected by & and cause an outgoing service failure.



(a) Outgoing value failure (b) Outgoing service failure

Figure 4.16: TFPGs with identical incoming failure variables

However, the OR-connection also covers the case that both incoming failures occur at the same time, even though this would cause an outgoing service failure. As a consequence, the correct TFPG for the outgoing value failure would connect both incoming failures by XOR as shown in Figure 4.17.

However, TFPGs with XOR-nodes cannot be analyzed by AShOp, because XOR cannot be mapped to Petri nets. We know that

$$a \oplus b = (\neg a \wedge b) \vee (a \wedge \neg b)$$

The mapping of OR-nodes and AND-nodes to a TPN was shown in Def. 3.4.4. $\neg a$ however cannot be modeled by a Petri net [EWM90, EW94]. Consequently, XOR cannot be modeled by a Petri net or TPN.

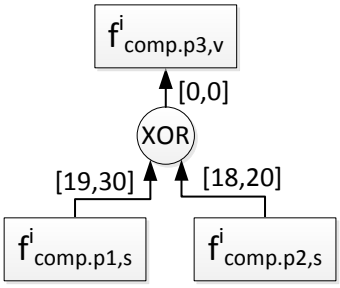


Figure 4.17: Corrected TFPG of the TFPG of Figure 4.16(a)

Moreover, negative events are usually not considered in failure propagation models, because these events are hard to figure out during manual construction. Negations and negative events are only introduced during the automatic generation. Here, the complete state-based behavior is taken into account and the generated failure propagation is finer.

We therefore do not replace OR-nodes by XOR-nodes. Consequently, the results of AShOp will be more pessimistic but safe. This means, the computed occurrence probabilities of hazards may be higher than the exact occurrence probabilities but never lower. This is because we do not reduce the set of valid combinations but extend it.

If none or only one of the input variables are true, the results are the same for both operators. OR becomes true if at least two variables are true at the same time. XOR will be false for this case. Thus, OR is true for a larger set of events. The probability of OR becoming true is always higher than the probability of XOR becoming true[3]

4.4 Summary

We have presented an approach for the automatic generation of timed failure propagation graphs (TFPGs) from timed automata. The automatic generation of TFPGs relieves the developer from the manual construction of TFPGs where especially the manual estimation of propagation times is difficult.

We particularly focus on the identification of the failure classes service, timing, and value and the computation of propagation times. Propagation times are computed from the real-time statecharts (cf. Section 2.2.2) of MechatronicUML component types. However, any state-based model, which is compatible to timed automata, may be used. We analyze deviations in the control flow to identify timing and service failures. For this, we compare the reachable behavior of a component type with and without injected failures. Value failures are identified by the data flow, which we analyze by an existing approach for slicing on extended finite state machines [ACH+12].

[3]This only holds for non-negative events.

The generated TFPGs are used by AShOp to analyze how self-healing operations affect the propagation of failures in the system. In particular, TFPGs allow for computing where failures are located in the system at the time when a self-healing operation is applied.

5 Analysis of Self-healing Operations

In this chapter, we present our approach for the analysis of self-healing operations (AShOp) as published in [PST11, PST13]. AShOp checks for a given self-healing operation whether it is executed fast enough to reduce the occurrence probability of a hazard such that the occurrence probability becomes acceptable. AShOp uses the TFPGs, which have been generated from the behavior models of the system components as described in the previous chapter.

There exist three approaches that analyze hazard probabilities in reconfigurable systems. The approach of Walter et al. [WGR+09] computes the probability that a system fails despite the execution of self-healing operations. However, it does not analyze how the self-healing operations affect the system. The approach of Giese et al. [GT06] computes the hazard occurrence probabilities of all possible architectures of a reconfigurable system. This approach computes hazard occurrence probabilities for system architectures. However, it does not consider the location of failures in the system at the time when the architecture is changed. Consequently, the approach is not able to analyze how the reconfiguration affects the propagation of failures, which already occurred. The approach of Güdemann et al. [GOR06] checks whether the system can always be returned into a safe state once a hazard has occurred. However, this approach does not take propagation times or the duration of the reconfiguration into account. It is thus not able to check whether a self-healing operation is executed fast enough.

To overcome the limitations mentioned above, we extend the component-based hazard analysis of Giese et al.[GT06]. Our extension of the component-based hazard analysis [GT06] comprises two parts. First, we take the propagation times of failures into account. The propagation times are needed to analyze how far failures propagate within a specific time span. Second, we analyze the effect of the structural reconfiguration on the propagation of failures. This is needed to check whether failures are removed from the system by removing components or whether failures are stopped from propagating further by removing connections between components. The combination of both parts allows for analyzing whether a self-healing operation is executed fast enough.

In contrast to the existing approaches [GOR06, GT06, WGR+09], AShOp uses the information provided by the behavior models of the components to generate TFPGs. On the one hand, they contain all information of the behavior models which are relevant for analyzing failure propagation. On the other hand, the benefit of TFPGs is their minimality concerning the information needed for the analysis of propagation times of failures. The generation from the behavior

models allows considering exactly that part of the behavior, which is necessary to analyze the failure propagation. This makes the analysis more efficient and allows thus for analyzing larger systems.

In contrast to the approaches of Güdemann et al. [GOR06] and Walter et al. [WGR+09], our analysis uses structural reconfiguration. Consequently, it benefits from the advantages of component-based development, particularly reuse and facilitated maintainability.

AShOp checks for a hazard and a self-healing operation, whether the self-healing operation reduces the occurrence probability of the hazard such that the hazard becomes acceptable. Therefore, AShOp analyzes which MCS of the hazard are critical after the application of the self-healing operation. An MCS is called critical if the self-healing operation cannot prevent the errors of the MCS from causing the hazard. The occurrence probabilities of the critical MCSs provide the information about the acceptability of the hazard after the application of the self-healing operation: The occurrence probability of the hazard is acceptable, if the joint occurrence probabilities of the critical MCS are acceptable. In this case, the self-healing operation is successful.

AShOp is executed in several steps that are illustrated in Figure 5.1. The steps are depicted by rounded rectangles and objects are represented by sketches of the objects. The control flow is given by the enumeration of the activities. The gray arrows depict the object flow.

Each step that is shown in Figure 5.1 is executed automatically. The developer provides a hazard, a threshold for an acceptable hazard occurrence probability, the self-healing operation, the deployment diagram, and the real-time statecharts of the component instances in the deployment diagram. For hardware nodes of the deployment diagram, either the behavior is modeled by real-time statecharts such that TFPGs are generated or the TFPGs of the hardware nodes are given.

First, (1) the TFPGs are generated from the real-time statecharts of the component types, which are instantiated in the deployment. Optionally, TFPGs are generated from real-time statecharts of hardware nodes. The result is the TFPG of the deployment diagram. The generation of TFPGs will be explained in full detail in Chapter 4.

The generated TFPGs are used to compute the MCSs and the hazard probabilities by the hazard analysis of Giese et al. [GT06]. The TFPGs are therefore translated into the failure propagation models of Giese et al. [GT06]. The timing annotations of TFPGs are thereby ignored. Based on the computed hazard probabilities and the deployment diagram, the developer constructs a self-healing operation. This is the only manual step. We omit these steps from Figure 5.1, because they are not in the focus of this thesis.

The next four steps (Steps 2 to 5) are conducted for each MCS that has been computed by the hazard analysis.

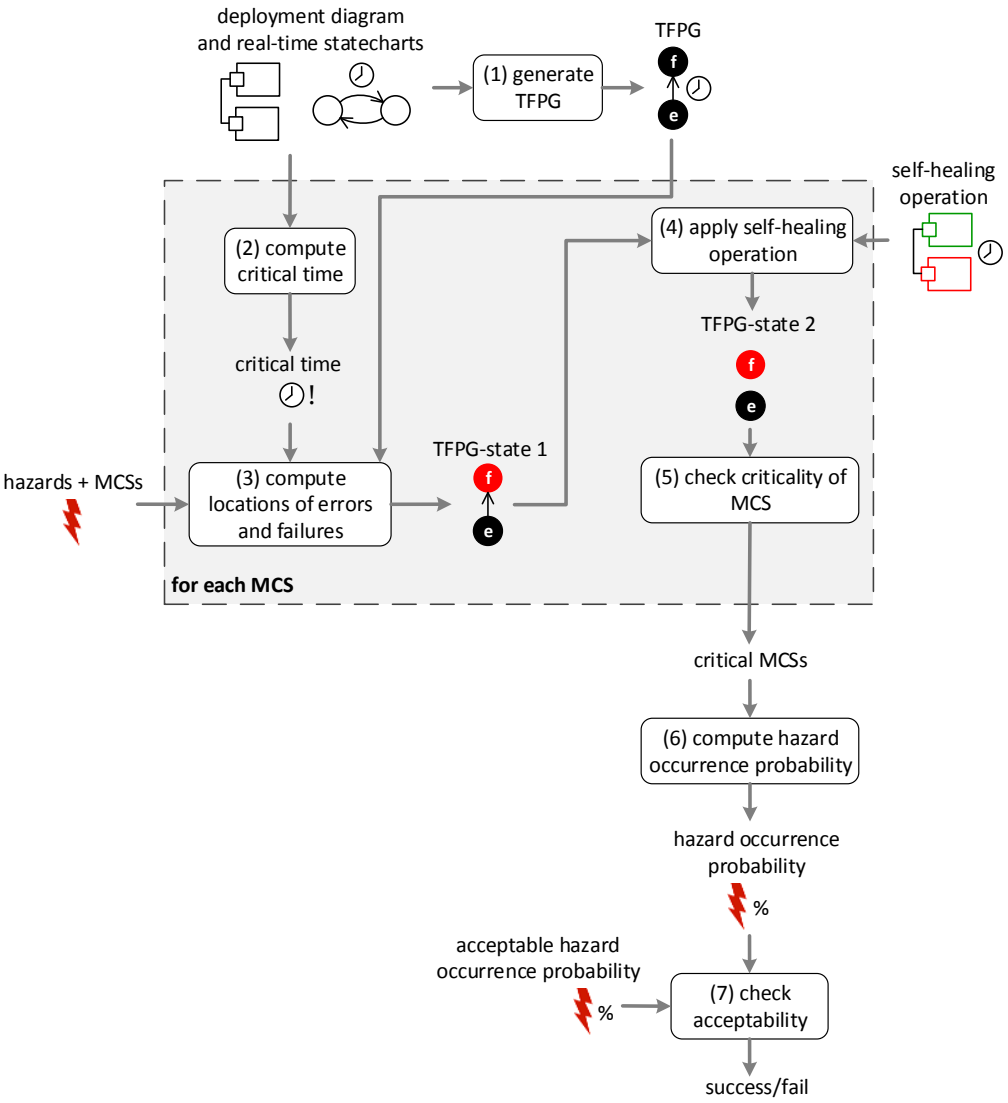


Figure 5.1: AShOp process

AShOp (2) computes the *critical time* which is the maximum time span between the detection of the error or failure and the completion of the self-healing operation. The critical time may vary due to system specific properties as has been explained in Section 2.2.4. We take the maximum value to analyze the worst case: the failures propagate as far as possible. This guarantees that critical failures can always be stopped, even in the case that the execution of the self-healing operation needs the maximum time that is specified in the real-time statecharts and TCSDs.

Next, AShOp (3) computes the locations of the errors and failures in the system at the point in time when the structural reconfiguration is applied. Based on the MCSs, it analyzes how far failures propagate through the system during the critical time. The result is the state of the TFPG (TFPG-state 1) that

contains all error and failure variables that are reachable before the execution of the self-healing operation.

The naive way of the described computation would be to perform the reachability analysis on the TPN of the whole system. But reachability analysis on TPNs [CR05] is equivalent to timed model checking which is exponential on the number of states and clocks [Alu99]. We improve the performance and thus the feasibility of AShOp by performing the reachability analysis only on the part of the system that is affected by the self-healing operation.

After the reachability analysis, (4) the self-healing operation is applied. It changes the structure of the deployment diagram and thereby the structure of the TFPG. The self-healing operation may remove errors and failures from the system or cut off propagation paths along which failures propagate to the hazard. The result of this step is the state of the TFPG of the reconfigured deployment diagram (TFPG-state 2) that contains all errors and failures that remain in the system after the application of the self-healing operation.

In the next step, the (5) criticality of the MCS is checked based on TFPG-state 2. This means, AShOp checks whether the errors and failures, which remain in the system after the application of the self-healing operation, still lead to the hazard. If this is the case, the MCS is critical. Steps (2) to (5) are repeated for each remaining MCS, which has not yet been analyzed.

It may happen that another reconfiguration is triggered before the self-healing operation is executed. In this case, this reconfiguration needs to be analyzed as well during Step two to five.

The (6) occurrence probability of the hazard is computed based on the occurrence probabilities of the critical MCSs. If the occurrence probability is acceptable (7), the self-healing operation is successful in reducing the hazard. The developer continues with generating program code. If the occurrence probability of the remaining MCSs is not acceptable, the self-healing operations have to be revised and analyzed again.

The errors in hardware components may be detected at runtime using for example model-based fault diagnosis [SFP02]. However, an error cannot be observed directly. The detection observes a failure at the port of the hardware component or at the port of another component. The errors, which cause the failure, and an appropriate self-healing operation are stored in a fault dictionary [PR92]. This fault dictionary associates failures in the systems with sets of errors that cause these failures (cf. Section 2.1.3). The fault dictionary is extended by self-healing operations, such that the fault detection not only identifies the causes of failure but also triggers a self-healing operation.

5.1 Example

In this chapter, we analyze the self-healing operation which has been introduced in Figure 3.2 of Chapter 3. This self-healing operation is applied to the configu-

ration of the speed control subsystem of the RailCab (Figure 3.1 of Chapter 3) if an error in the distance sensor occurs, because this error would lead to the hazard wrong_speed. The occurrence probability of this hazard is 0.27829 as we have computed in Section 3.4.3. The threshold for the occurrence probability of the hazard wrong_speed is 0.2. The MCSs of the hazard wrong_speed are $\{e_{s1,v}, e_{s2,v}\}$, $\{e_{gps,v}\}$, $\{e_{dr,v}\}$, and $\{e_{wlan,v}\}$. In this chapter, we use the MCS $\{e_{dr,v}\}$ to illustrate the execution of Steps (2) to (5).

5.2 Computing the Critical Time

The critical time is the time during which a failure propagates through the system after it has been detected and before the self-healing operation is completed. We use the critical time to compute the locations of errors and failures in the system at the time when the self-healing operation is executed.

By taking the maximum time span between the error occurrence and the completion of the self-healing operation, we consider the worst case: we let the failures propagate as far as possible through the system. This is necessary, because we do not know which step of the self-healing operation is executed at which time. Consequently, we let the failures propagate for the maximum duration of the self-healing operation and apply all steps of the self-healing operation in zero time. This is a safe approximation, because failures will not propagate for a longer time than the maximum duration of the self-healing operation. However, they may have less time and thus propagate a shorter distance than the distance, which was computed during the analysis.

The critical time is composed of three parts (cf. Figure 5.2): the *error delay*, the *reconfiguration delay*, and the *duration of the self-healing operation*. The error delay is the time between the occurrence of the errors of the MCS and the failure, which is detected by the failure detection. The reconfiguration delay is the time between the detection of the failure and the start of the self-healing operation. In most cases, the self-healing operation is not executed immediately after the failure has been detected. The information that a self-healing operation has to be applied must be transmitted to the component where the self-healing operation is executed. This transmission results in the reconfiguration delay. The error delay, the reconfiguration delay, and the duration of the self-healing operation follow one another directly as depicted in Figure 5.2. The critical time is consequently the sum of these three parts.

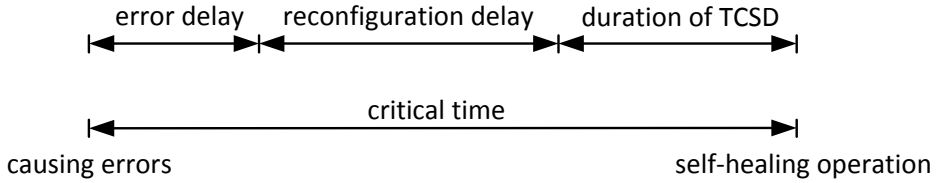


Figure 5.2: Composition of the critical time

5.2.1 Error Delay

The error delay is computed from the propagation times of the paths in the TFPG. A depth first search identifies all paths that lead from the detected failure back to its causing errors. Then, the propagation times of all these paths are computed. The error delay is the maximum value over the propagation times of all these paths. By taking the maximum value, we consider the case, that the failure is detected at the latest time possible. The error delay is stored in the fault dictionary. It complements the relation between error and failure.

The value error $e_{dr,v}$ in the distance sensor dr:DSensor of Fig. 3.8 is for example detected by the incoming failure $f^i_{sa.p1,v}$ at the incoming port of the component instance sa:Sanity. The error delay is thus the maximum time span between the occurrences of $e_{dr,v}$ and the incoming failure $f^i_{sa.p1,v}$ at sa:Sanity. The propagation time interval for the path between these nodes is $[1 + 5, 2 + 6] = [6, 8]$. Thus, the error delay is 8 time units.

5.2.2 Reconfiguration Delay

The reconfiguration delay is computed from the runtimes of paths in the reachable behavior of the system as explained in Section 4.2.1.

We extract all paths from the reachable behavior that contain the message of the detection of the error or failure and the execution of the self-healing operation. The detection messages and the according self-healing operations are looked up in the fault dictionary as explained in Section 2.1.3. We compute the maximum delay for each path as described in Section 4.2.1. The reconfiguration delay is the delay of the path with the highest maximum delay.

In our example, the failure detection detects the incoming value failure at the component instance sa:Sanity (cf. Figure 3.8). This value failure enters sa:Sanity via the message sa-p1.dist. For the reconfiguration delay, we thus need to compute the runtime from the receiving of sa-p1.dist to the firing of the transition with the side effect DiscDistSensor() using the reachable behavior of Figure 3.6. The delay between the receipt of the value of the distance sensor dr:DSensor in the timed automaton of Figure 3.3 and the final trigger of the TCSD by the execution of the side effect in the timed automaton of Figure 3.4 is $[35, 53]$.

In our example, the maximum duration of the TCSD is 86 time units. Therefore, the critical time is computed by $8 + 53 + 86 = 147$.

5.3 Compute Locations of Errors and Failures

The locations of errors and failures are computed by a reachability analysis on the underlying TPN of the TFPG (cf. Section 3.4) of the system. For the reachability analysis on TPNs, we use the reachability analysis of Cassez and

Roux [CR05]. The input for the computation of the locations of errors and failures is a TFPG. In this TFPG, all errors and failures that are contained in the MCS are activated. The reachability analysis simulates the propagation of failures through the TFPG by means of a TPN for the time span, which is specified by the critical time. The result is the state of the TFPG (cf. Def. 3.4.5) after the critical time. This means, all error and failure variables of the TFPG that were visited during the reachability analysis are active.

The locations of errors and failures are computed only on the part of the system that is affected by the self-healing operation. In the deployment diagram, we call this part the *affected subgraph*. By reducing the number of places of the TPN that considered by the reachability analysis we also reduce the number of clocks that have to be analyzed. Of course, the runtime remains exponential. However, the approach becomes applicable in practice. This is shown by an evaluation of our implementation in Section 7.3.3.

Definition 5.3.1 (Affected Subgraph of a TFPG)
Let $w = (G_{conf} = (V_{conf}, E_{conf}, s_{conf}, t_{conf}), type)$ *be a deployment diagram with* $V_{conf} = \overline{K} \cup H \cup \overline{P}_K \cup P_H$ *and* $r = (LHS, RHS, d)$ *the TCSP of the TCSD that specifies the self-healing operation.* r *is matched into* G_{conf} *by a matching* m*. Let further* $G = (V, E, f_s, f_t, I, l, \iota, \eta)$ *be the TFPG of* w *over the error and failure specification* $\overline{\mathcal{V}}(G) = (w, \overline{\mathcal{E}}, \overline{\mathcal{F}}, \overline{f}_{\mathcal{E}}, \overline{f}_{\mathcal{F}})$*.*

Let $G_m = (\overline{\mathcal{V}}^{-1} \circ m)(LHS \setminus RHS)$ *and* $G_m = (V_m, E_m, f_{sm}, f_{tm}, I_m, l_m, \iota_m, \eta_m)$*. We define* V_m *the set of* affected nodes *of* G*.*

The affected subgraph G_A *of* G *is induced by the nodes of a set* X *of paths* $x = v_1, \ldots, v_k$*,* $v_1, ..., v_k \in V$*, where each* $x \in X$ *meets the following conditions:*

- $l(v_1) \in \overline{\mathcal{E}}$*,*
- $v_k \notin V_m$*,*
- $v_{k-1} \in V_m$*, and*
- $v_i \notin V_m$ *for* $i > k$*.*

The affected subgraph consists of the nodes which are changed by the TCSD that represents the self-healing operation (affected nodes), all paths that lead from error nodes to the affected nodes, and the direct successors of the affected nodes. The affected nodes are needed to assess the effect of the TCSD on the TFPG. The affected nodes are connected to all nodes of error variables that cause the failures, which are represented by the affected nodes. Thus, we can analyze whether failures may propagate to the affected nodes. The successors of the affected nodes are needed to analyze whether failures leave the part that is changed by the component story diagram. This information is used to analyze whether the failures, which remain in the system, may still cause the hazard.

The affected subgraph is computed from the union of the matchings of the LHS of all applicable TCSPs of the TCSD. This is sufficient for computing

the affected subgraph, because the order of the application of the TSCPs is not important for analyzing how the self-healing operation affects the failure propagation. Therefore, we consider TCSDs as sets of TCSPs in the definition above.

An example that illustrates the affected part of the TFPG of Figure 3.8 is shown in Figure 5.3. The affected subgraph is highlighted by a gray background. It consists of the three paths $e_{gps,v}, ..., OR$, $e_{dr,v}, ..., OR$ and $e_{wlan,v}, ..., OR$.

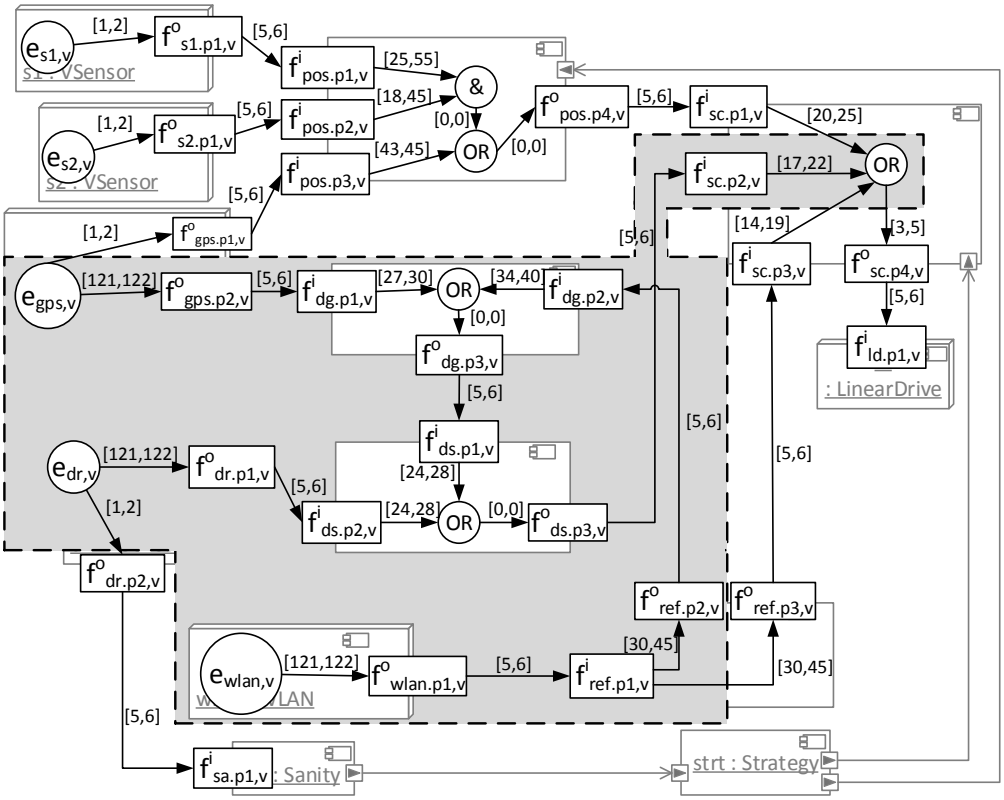


Figure 5.3: Affected subgraph

For computing the locations of errors and failures at the point of time when the self-healing is applied, we set the state of the affected subgraph according to the errors of the MCS. This means, the error variables that represent the errors of the MCS are set to "active". All other error and failure variables are set to "inactive". In our example, the state of the affected subgraph is set to $\{e_{dr,v}\}$.

We compute the state of the affected subgraph for the duration which is specified by the critical time (cf. Section 5.2). This state contains all error and failure variables that may be active before executing the TCSD completely.

Algorithm 4 formalizes the analysis of the affected subgraph.

Algorithm 4 first computes the tolerance time as explained in Section 5.2 (Line 1). Next, the affected subgraph of the TFPG is computed as defined in Definition 5.3.1 (Line 2) and the state of the TFPG is set as specified by

Algorithm 4 AnalyzeMCS

Require: $w = (G_{conf}, type)$ a deployment diagram,
$G = (V, E, f_s, f_t, I, l, \iota, \eta)$ the TFPG of the deployment diagram,
$\text{mcs} = \{e \in \mathcal{E}\}$,
r the TCSP of the TCSD that specifies the self-healing operation
Ensure: q'' the active error and failure variables after the reconfiguration for the lifetime of the system

1: $\text{tt} = computeToleranceTime(w, r)$
2: $G_A = (V_A, E_A, f_{sA}, f_{tA}, I_A, l_A, \iota_A, \eta_A) :=$ buildAffectedSubgraph(G, w, r)
3: $q(G_A) = setTFPGState(G_A, \text{mcs})$
4: $G_A :=$ computeReachableFailures (G_A, tt)
5: $w \overset{r}{\Rightarrow} \hat{w}$
6: $\hat{G}_A = \mathcal{V}^{-1}(\hat{w})$
7: return $q(\hat{G}_A)$

the MCS of the hazard. In the next step, the reachability analysis of Cassez and Roux [CR05] is applied to the affected subgraph (Line 4). The application is specified in more detail in Algorithm 5. It computes the state of the TFPG before the application of the self-healing operation. Then, the self-healing operation is applied (Line 5) and the TFPG is reconfigured (Line 6). Finally, the state of the reconfigured TFPG is returned (Line 7). The active error and failure variables correspond to the errors and failures that remain in the system after the application of the self-healing operation.

Algorithm 5 computeReachableFailures

Require: $G = (V, E, f_s, f_t, I, l, \iota, \eta)$ TFPG,
tt duration
Ensure: q the set of active error and failure variables for the duration tt

1: $\mathcal{T} = (P, T, \bullet(.), (.)\bullet, M_0, (\alpha, \beta)) = \mu(G)$
2: $\mathcal{M} =$ getReachableMarkings($\mathcal{T}$, tt)
3: **for all** $v \in V$ **do**
4: $\quad \eta(v) = inactive$
5: **end for**
6: **for all** $M \in \mathcal{M}$ **do**
7: $\quad$ **for** $i = 1$ to $|P|$ **do**
8: $\quad\quad v = \mu^{-1}(p_i)$
9: $\quad\quad$ **if** $m_i > 0$ **then**
10: $\quad\quad\quad \eta(v) =$ active
11: $\quad\quad$ **else**
12: $\quad\quad\quad \eta(v) =$ inactive
13: $\quad\quad$ **end if**
14: $\quad$ **end for**
15: **end for**
16: return G

Algorithm 5 computes the state of a TFPG for a given duration. For this, it computes the set $\mathcal{M}$ of reachable markings of the TPN for the duration tt using the approach of [CR05] (Line 2). The state of the TFPG G is reset in Lines 3 to 5. Then, the new state of G is set in Lines 6 to 5. Every node, whose place in the TPN contains at least one token, is set active (Line 10). All other nodes are set inactive (Line 12).

Figures 5.4 to 5.6 show states of the TPN and the TFPG of the affected subgraph during the reachability analysis. Figure 5.4 shows the state of the TPN $\mathcal{T}$ of the affected subgraph of Figure 5.3 at the point in time when the failure is detected. The error variable $e_{dr,v}$ has been activated, because this MCS is currently analyzed.

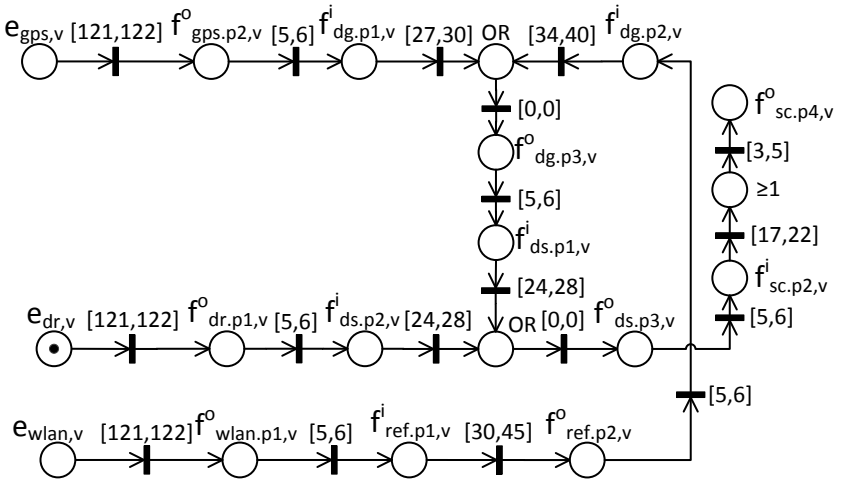


Figure 5.4: Marking of the TPN at the time when the error occurred

Figure 5.5 shows the reachable marking of the TPN that are reachable during the critical time. These markings are reachable at the end of the duration of the self-healing operation but before the execution of the TCSD. During the critical time of 147 time units, the failure variables $f^o_{dr.p1,v}$ and $f^i_{ds.p2,v}$ of the TPN are reachable. Their corresponding failures may thus have occurred in the real system.

Figure 5.6 shows the corresponding TFPG marked with the active error and failure variables of the TFPG-state that corresponds to the marked places in the TPN of Figure 5.5. The active error and failure variables reflect possible locations of errors and failures during the critical time.

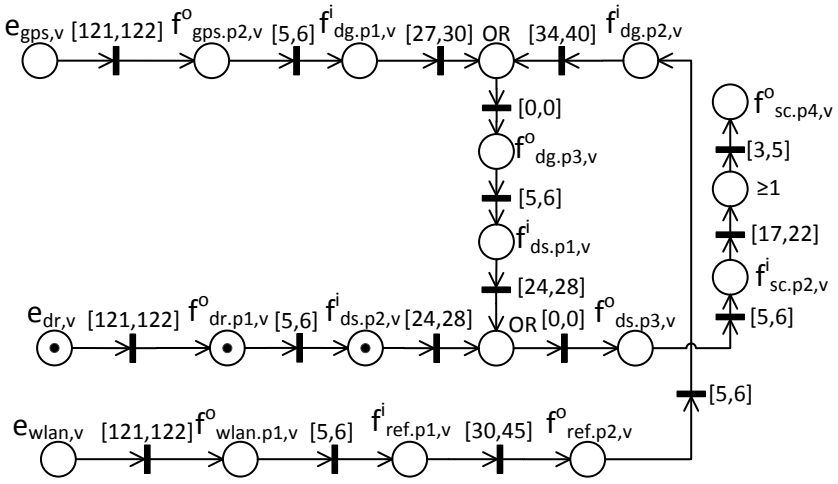


Figure 5.5: Reachable markings of the TPN during the critical time

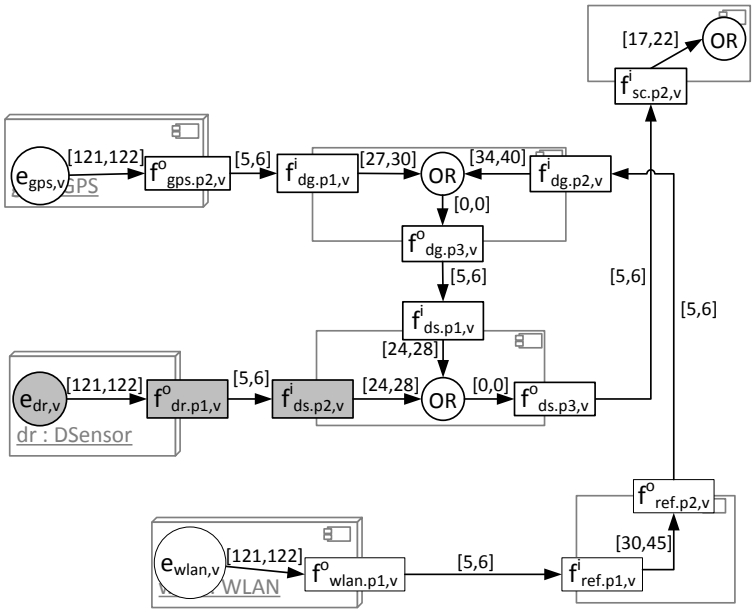


Figure 5.6: State of the affected subgraph after the critical time

5.4 Analyze the Criticality of the MCS

To analyze the criticality of the MCS m, we compute the MCSs M_r of the reconfigured TFPG using the component-based hazard analysis of Giese et al. [GT06] and check which error and failure variables of the MCSs M_r are active in the reconfigured TFPG. The MCS m is critical if at least one of the MCSs $m_r \in M_r$ contains only active error and failure variable. In this case, the errors and failures, which remain in the system, would still cause a hazard.

In a TFPG, only error variables, i.e., the first node of each path, which leads to the hazard, are candidates for the cause of a hazard. However, in the reconfigured TFPG, the paths between error and failure variables and the hazard may be interrupted. Consequently, the MCSs of the reconfigured TFPG may also contain failures. Moreover, there may be paths in the TFPG whose error node is inactive but subsequent failure nodes are active. In this case, the resulting MCS contains inactive errors, even though there are active failures, which still may cause the hazard.

We consequently modify the TFPG such that active nodes become the first nodes of each path that leads to the hazard. This is achieved by deleting the incoming edges of all active failure variables. In this way, only the active failure variable with the shortest distance to the outgoing failure that is part of the specified hazard remains connected to the hazard.

Figure 5.7 shows the TFPG of the speed control subsystem after the execution of the TCSD. Due to the application of the TCSD, the failure variables $f^i_{dg.p3,v}$ and $f^i_{sc.p2,v}$ and the edges between OR and $f^i_{ds.p1,v}$ and $f^o_{ds.p3,v}$ and $f^i_{sc.p2,v}$ have been removed. A new failure variable $f^i_{sc.p5,v}$ and an edge between $f^o_{dg.p4,v}$ and $f^i_{sc.p5,v}$ have been created. Further, the incoming edges of $f^o_{dr.p1,v}$ and $f^i_{ds.p2,v}$ have been removed to enable the computation of critical MCSs.

For the TFPG of Figure 5.7, we compute the MCSs $\{e_{s1,v}, e_{s2}\}$, $\{e_{gps,v}\}$, and $\{e_{wlan,v}\}$. The set of active error and failure variables is $\{e_{dr,v}, f^o_{dr.p1,v}, f^i_{ds.p2,v}\}$. None of the MCSs contains these variables. They are thus all non-critical and the hazard cannot occur anymore for the analyzed MCS $\{e_{dr,v}\}$. We repeat the steps described in Sections 5.2 to 5.4 to analyze the remaining MCSs of the hazard wrong speed. The result is the set of critical MCSs. An example of a MCS which is still critical after the self-healing operation of our example is $\{e_{s1,v}, e_{s2,v}\}$.

5.5 Analyze the Success of the Self-healing Operation

The occurrence probabilities of the hazard after the application of the self-healing operation is computed from the occurrence probabilities of the critical

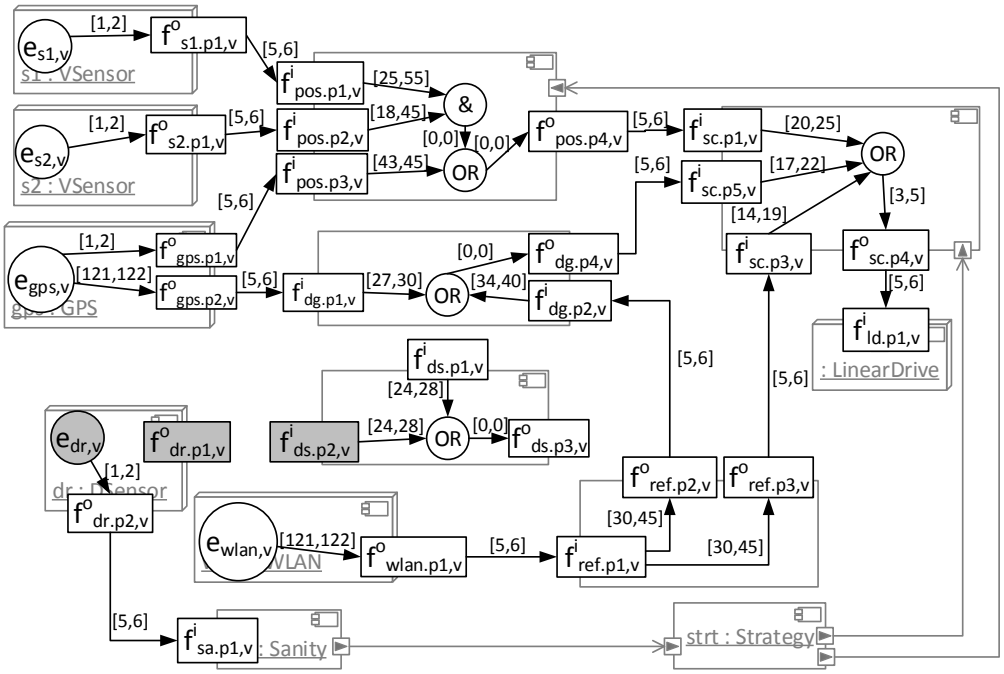


Figure 5.7: Reduced TFPG

MCSs. The critical MCSs are $\{e_{s1,v}, e_{s2,v}\}$, $\{e_{gps,v}\}$, $\{e_{wlan,v}\}$. The resulting occurrence probability is computed by

$$\begin{aligned} & 1 - ((1 - p(e_{s1,v}) \cdot p(e_{s2,v}))(1 - p(e_{gps,v}))(1 - p(e_{wlan,v}))) \\ =& 1 - ((1 - 0.01)(1 - 0.1)(1 - 0.1)) \\ =& 0.1981 \end{aligned}$$

The occurrence probability 0.1981 is smaller than the threshold of 0.2. The occurrence probability of the hazard wrong speed is thus acceptable. We conclude that the self-healing operation is successful.

5.6 Remarks

AShOp handles multiple errors as well as single errors. This is guaranteed by applying AShOp to *MCSs* instead of single errors.

It cannot happen that paths are created by the TCSD that build a "bridge" that lets a failure slip through the self-healing. Hence, it does not occur that AShOp yields that the failure can be stopped though it is not. This is due to the semantics of TCSDs: First, all objects that are to be destroyed, are removed. Then, all objects that are to be created are created.

Cycles in the TFPG can also be handled by AShOp, because we map our TFPG to a TPN. The reachability analysis of Cassez and Roux [CR05] handles cycles in TPNs. Further, the hazard analysis of Giese et al. [GT06] handles cycles, as well.

5.7 Summary

In this chapter, we presented an approach for the analysis of self-healing operations. With this analysis, the system developer is able to analyze whether a self-healing operation reduces a hazard such that it becomes acceptable. In contrast to the approach of Nafz et al. [NSS+11], our approach also checks whether the self-healing is executed fast enough to reduce the hazard occurrence probability.

In order to check whether the self-healing operations reduce the occurrence probability of a hazard, AShOp checks for each MCS of a hazard whether failures are stopped from propagating before the hazard occurs. Therefore, AShOp first checks how far failures propagate between their detection and the completion of the self-healing operation. AShOp then determines how the self-healing operation affects the failure propagation at the time of the completion of the TCSD. Finally, it checks whether the errors and failures that remain in the system after the application of the self-healing operation still cause the hazard. For all MCSs that still lead to the hazard, AShOp computes the occurrence probability after the application of the self-healing operation. If the occurrence probability of all remaining MCSs is acceptable, the self-healing operation is successful in reducing the hazard. If the occurrence probability of the remaining MCSs is not acceptable, the self-healing operations have to be revised and analyzed again. AShOp becomes applicable by performing the reachability analysis only on that part of the system that is affected by the self-healing operation.

The analysis, which we presented in this chapter, is applied at design time. This implies that all configurations of the system are known at design time, because the configuration is needed to analyze the failure propagation between components. However, in reconfigurable systems, configurations may be created at runtime which have been unknown at design time. Therefore, we developed a framework that allows for executing AShOp at runtime. This is the subject of the next chapter.

6 Analysis of Self-healing Operations at Runtime

The system developer has to guarantee acceptable hazard occurrence probabilities for all possible system architectures at design time (cf. Section 2.4). The configuration of a system must be known in order to analyze how a self-healing operation affects the propagation of failure through a system. However, in reconfigurable systems, it is possible that system architectures occur only at runtime and are unknown at design time. This applies particularly to systems of systems, where different systems are combined at runtime to provide functionalities that none of the systems could provide alone [CK10].

In this chapter, we call configurations where the self-healing operations of the system do not reduce hazards occurrence probabilities below the required thresholds *unsafe configurations*. For configurations, which only occur at runtime, we use the term *unknown configuration*. We will call the system of systems *system* and a system which is part of the system of systems *subsystem*

An example of such a system of systems is the RailCab convoy (cf. Section 1.1). In the future, there may exist smart rail vehicles apart from the RailCab, which are produced by foreign manufacturers. Of course, it would be useful, if these RailCabs and the foreign vehicles formed a convoy. For this, they need to establish a communication connection. This connection leads to an unknown configuration, because the system architecture of the foreign vehicle was, of course, unknown to the developers of the RailCab. Consequently, the effect of self-healing operations could not be computed at design time either and there may appear unsafe configurations at runtime. To guarantee that no unsafe configurations occur, the effect of self-healing operations on unknown configurations must be analyzed at runtime [CGKM12, SBT11].

There exist several approaches that analyze safety properties at runtime. However, only one approach analyzes the safety requirements of systems before they are connected [SBT11]. Yet, is does not take self-healing operations into account. The main focus of runtime analysis approaches is the detection of anomalies in the executed system behavior and to return the system back to its intended behavior [DDK+07, FGT11, GMS12, KMM07, Rus08, SRA04]. However, they do not analyze self-healing operations.

To analyze self-healing operations at runtime, we construct reachable configurations at runtime and perform AShOp. Reconfiguration rules that lead to unsafe

configurations are locked. Consequently, unsafe configurations are not reachable. We published an approach in [PT09, PHST12] that uses this framework for analyzing risk at runtime.

Figure 6.1 shows an overview of the execution of AShOp at runtime. In the beginning, (1) the reachable configurations are constructed. This is done by the reachability analysis of Suck et al. [SHS11]. It is the only approach, which takes behavior and reconfiguration into account and considers time. Consequently, only those reconfiguration rules are applied which are reachable as side effects of the real-time behavior. In contrast to other approaches, e.g, [dLGB+10, MdR07, ÖM07, RDV09], that apply each reconfiguration rule to each reachable architecture, the space of reachable architectures is reduced significantly.

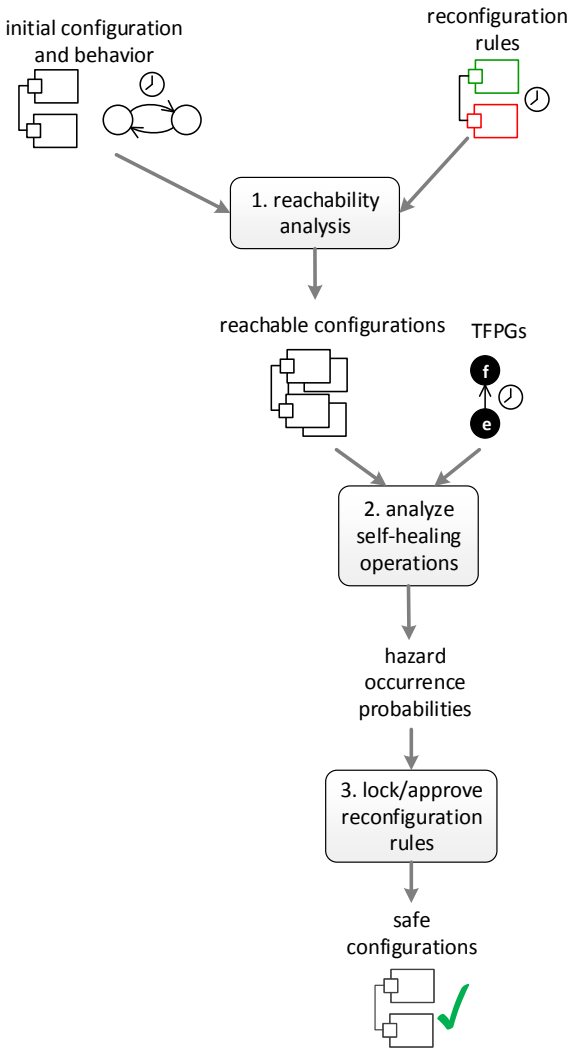


Figure 6.1: Overview of the runtime analysis

In the next step, (2) the self-healing operations are analyzed as described in Chapter 5. Of course, all models required for hazard analysis and reachability analysis must be stored in the running system and must be available at runtime.

In the last step, (3) reconfiguration rules that lead to unsafe configurations are locked. In the end, only safe system architectures are reachable.

6.1 Example

We illustrate the execution of AShOp at runtime by the example of the speed control subsystem of the RailCab which has been introduced in Section 3.1. We extend this example by a rail vehicle of a foreign manufacturer that meets the RailCab on the track. The vehicles try to build a convoy and thus need to establish a communication connection.

Figure 6.2 shows an excerpt of the deployment diagram of a rail vehicle of a foreign manufacturer. This deployment diagram is unknown to the RailCab developers at design time.

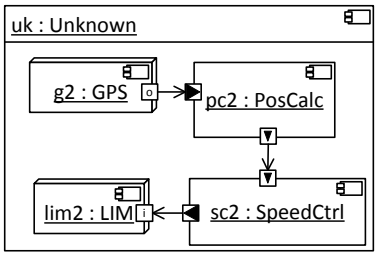


Figure 6.2: Deployment diagram of the foreign rail vehicle

Like the RailCab, the foreign vehicle drives with a linear drive lim2:LIM. The electric current, which is applied to move the vehicle, is controlled by the speed controller s2:SpeedCtrl. The target speed is computed from the target position of the vehicle on the track, which is provided by pc2:PosCalc. The foreign vehicle measures speed by a GPS-sensor g2:GPS.

The TFPG of the foreign vehicle in Figure 6.3 contains the error $e_{g2,v}$ located in the component instance gps:GPS. This error may lead to the outgoing value failure $f^{o}_{sc2.p2,v}$ of the speed controller and cause a wrong speed of the unknown vehicle.

Once the vehicles have met on the track, they need to establish a connection in order to build a convoy. Figure 6.4 shows the reconfiguration rules that create the necessary components for convoy mode. The time intervals of the reconfiguration rules are omitted. The reconfiguration rule createCoordinatorCtrl() of Figure 6.4(a) creates a component instance of type Coordination and connects it to a component instance of type PosCalc. The newly created component instance is also connected to a newly created multi-port, which establishes a communication link to the member vehicles of the convoy. The component instance co:Coordination coordinates the convoy by sending reference and target parameters, e.g., target speed, to the convoy members.

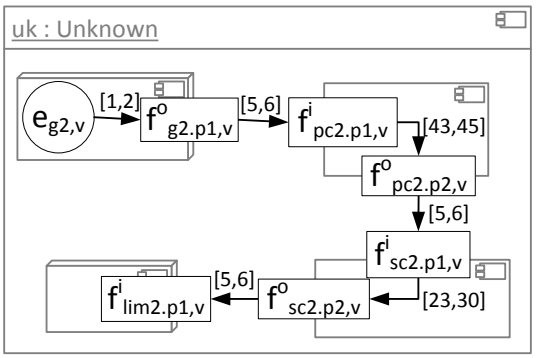
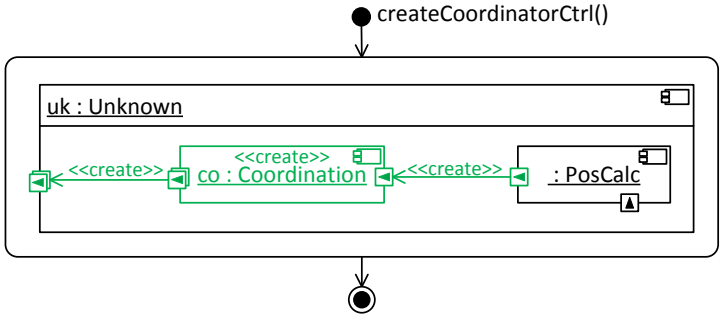
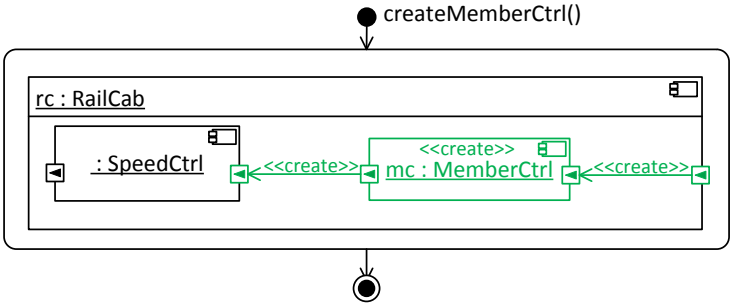


Figure 6.3: TFPG of the foreign rail vehicle



(a) Convoy coordinator



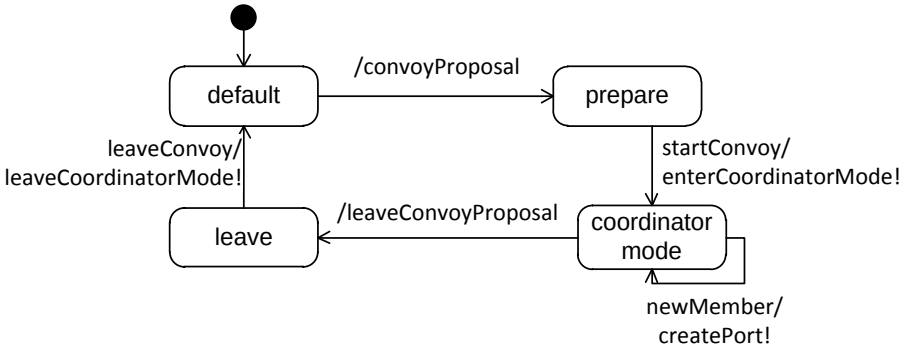
(b) Convoy member

Figure 6.4: TCSDs for establishing a convoy

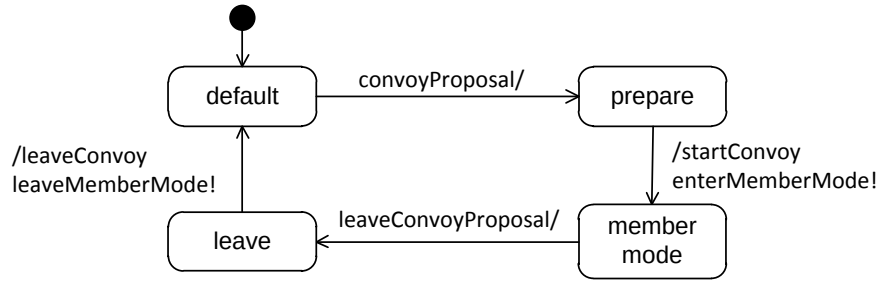
The reconfiguration rule createMemberCtrl() of Figure 6.4(b) creates a component instance of type MemberCtrl and connects it to the speed controller SpeedCtrl. The component instance mc:MemberCtrl processes the reference data provided by the convoy coordinator. This newly created component instance is further connected to a single port, which establishes a connection to the convoy coordinator.

Before the convoy is built, the vehicles need to agree on the convoy and their roles, i.e., either coordinator or member. Figure 6.5 shows the according communication needed to build a convoy between the two rail vehicles of our example. The communication is simplified with the intention to only show the

steps necessary to trigger the required reconfigurations. Figure 6.5(a) shows the role statechart of the unknown rail vehicle. Figure 6.5(b) shows the role statechart of the RailCab. The unknown rail vehicle proposes to build a convoy by sending the asynchronous message convoyProposal (cf. Figure 6.5(a)). The RailCab receives the message convoyProposal, reacts by sending startConvoy, and switches to member mode (cf. Figure 6.5(b)). This message is received by the unknown rail vehicle. It switches to coordinator mode. Both transitions are labeled with synchronizations that trigger the reconfigurations required for convoy mode. The synchronization enterCoordinatorMode of the unknown vehicle triggers the initial transitions of the adaptation statechart of Figure 6.6. The synchronization enterMemberMode of the RailCab triggers the initial transition of the port statechart of Figure 6.7.



(a) Port statechart of the unknown rail vehicle



(b) Port statechart of the RailCab

Figure 6.5: Communication for building and leaving a convoy

Figure 6.6 shows the adaptation statechart of the multi-role Coordinator. The initial transition is triggered by the synchronization enterCoordinatorMode of the statechart of Figure 6.5(a). The transition triggers the side effect createCoordinatorCtrl. This side effect (cf. Figure 6.4(a)) creates the components and communication links required to act as a coordinator and communicate to one convoy member. If the synchronization createPort is received from the internal behavior, the side effect addMember is executed. It creates another subport for the coordinator to communicate with another convoy member. If the synchronization leaveCoordinatorMode is received from the internal behavior, the side effect destroyCoordinatorCtrl destroys all components and communication links required for coordinating a convoy.

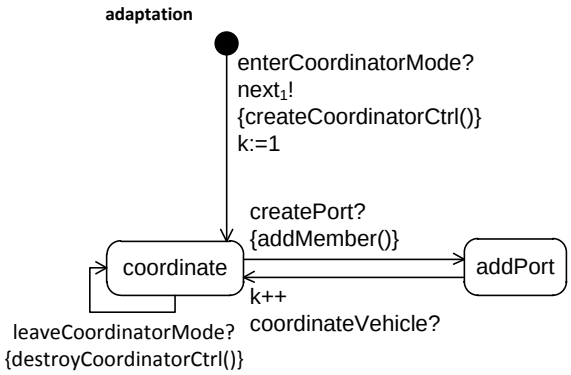


Figure 6.6: Behavior of the multi-role Coordinator

The variable k stores the number of coordinator subports which have been created and thus the number of vehicles which are connected to the coordinator. The message $\mathsf{next}_1!$ at the initial transition triggers the first subrole of the multi-role. We implement synchronization between different subroles and between subroles and the adaptation statechart by using integers. Synchronizations have the form si with s being the synchronization's name and i the index. Only synchronizations with the same index synchronize.

Figure 6.7 shows the port statechart of the single role Member. The initial transition is triggered by the synchronization enterMemberMode of the statechart of Figure 6.5(b). The side effect createMemberCtrl (cf. Figure 6.4(b)) of the initial transition creates the components and communication links required to act as a convoy member. If the synchronization leaveMemberMode is received from the internal behavior, the side effect destroyMemberCtrl destroys all components and communication links required to act as a member in a convoy.

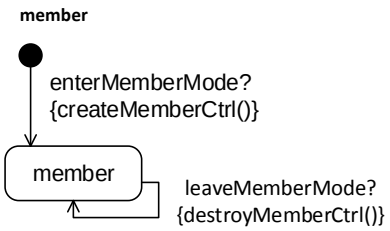


Figure 6.7: Behavior of the single role Member

6.2 System Extensions

In order to execute AShOp at runtime and lock reconfiguration rules, each subsystem needs to be extended by a *safety manager* that implements this functionality.

AShOp is executed globally for the whole system. It is therefore only implemented by the safety manager of one subsystem. This safety manager is extended by a so-called *analyzer* that executes AShOp. Therefore, the analyzer constructs the configuration of the whole system and performs the reachability analysis before executing AShOp. The locking of reconfiguration rules has to be performed in each subsystem, because only the subsystems can control their behavior. Therefore, the safety manager of each subsystem contains a *reconfiguration controller* that evaluates the data of the *analyzer* and locks reconfiguration rules if they lead to unsafe configurations.

The analyzer may be deployed on any subsystem. Leader election protocols [TvS08] can be employed to determine at runtime which component will execute AShOp. In systems where the computing hardware resources are limited, the analyzer may be deployed on an external unit as for example a desktop computer.

Figure 6.8 shows an excerpt of the deployment diagram of the RailCab and the unknown rail vehicle driving in a convoy. The software of the two vehicles has been extended by a *safety manager* that is implemented by the component type SafetyManager. The safety manager of each subsystem contains an instance of the component type ReconfCtrl that implements the reconfiguration controller. The safety manager of the RailCab contains an instance of the component type Analyzer that implements the analyzer.

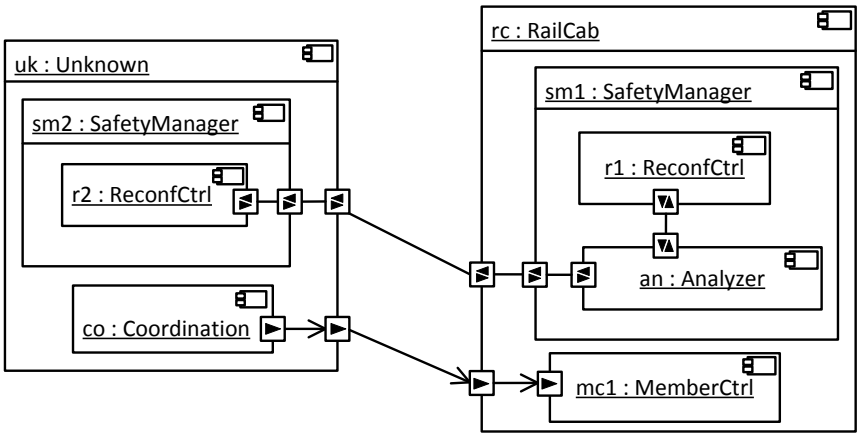


Figure 6.8: Subsystems with safety manager components

We maintain a separation of concerns by encapsulating the functions of AShOp into separate components, i.e., we do not mix the system functions with the analysis [MSKC04].

6.2.1 Analyzer

For AShOp, all subsystems send their current configuration, reconfiguration rules, and failure propagation models to the analyzer. The analyzer constructs the system configuration and computes the reachable system configurations for a fixed number of steps. It analyzes the self-healing operations of each subsystem for each reachable system configuration and decides whether the configuration is a safe configuration. Subsequently, the analyzer broadcasts the result to the safety manager of the subsystem, which executes the reconfiguration rule that leads to the analyzed configuration. After a subsystem has executed a reconfiguration, it informs the reconfiguration controller about the executed reconfiguration.

Construction of the System Configuration

For the construction of the system configuration, the analyzer needs to know, which components are part of the system and how the components are connected. Each time a subsystem wants to join the system, e.g., a vehicle intends to enter the convoy, the analyzer needs to be informed about which connections this component intends to establish to other components. Therefore, the analyzer assigns IDs to all subsystems. The connected subsystems exchange their IDs and transmit the IDs of subsystems they are connected with the analyzer. Using these IDs, the analyzer reconstructs which subsystems are connected and thereby the configuration of the system.

A new subsystem, that intends to join the system, requests an ID from the analyzer and from the subsystems, it wishes to connect with. This information is transmitted to the analyzer that judges whether the resulting hazard occurrence probability of the system is acceptable or not. The communication between a new subsystem and the system is shown in Figure 6.9.

In this scenario, a third vehicle, namely nr:RailCab, wants to join the convoy of the RailCab and the rail vehicle of the foreign manufacturer. Therefore, the new vehicle first requests the ID of the vehicle it wants to establish a connection with, namely uk:Unknown. Afterwards, nr:RailCab sends its request to entry to rc:RailCab that acts as the analyzer. The analyzer either permits the connection or rejects the request. If the connection is allowed, both vehicles connect and nr:RailCab stores the ID of uk:Unknown for further reconfiguration requests.

Reachability Analysis

For computing the reachable system configurations, we perform a reachability analysis on the reconfiguration behavior. As described in Section 2.2, the reconfiguration behavior consists of real-time statecharts that execute reconfiguration rules as side effects of their transitions. We compute the reachable configurations using the reachability analysis of Suck et al. [SHS11]. Based on

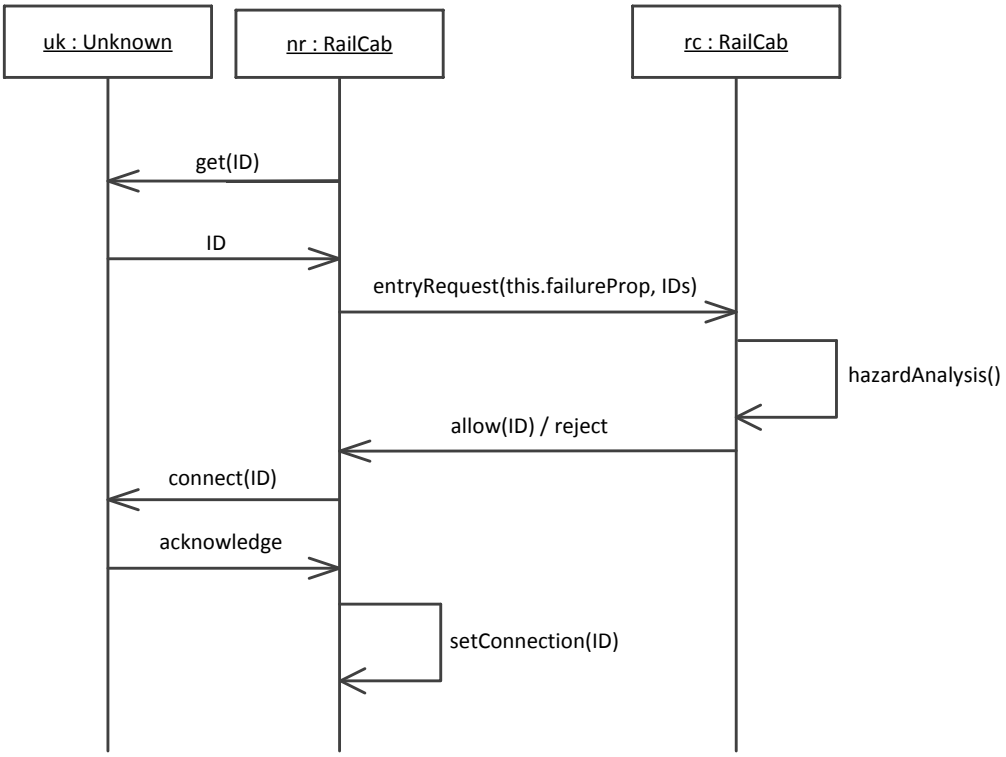


Figure 6.9: Communication for a new vehicle connecting to the system

the current configuration and the set of reconfiguration rules, the reachability analysis computes all possible successive configurations. For this, the reachability analysis also considers the real-time statecharts of the system: Only those reconfiguration rules are applied that are reachable as side effects from the current real-time state. The result is a labeled transition system whose states are configurations and whose transitions correspond to applications of reconfiguration rules. The transitions are labeled with the names of the applied reconfiguration rules.

Figure 6.10 shows the labeled transition system that represents the reachable configurations when building a convoy. The labeled transition system consists of four configurations, namely d1, d2, d3, and d4. We only show the parts of the configurations that are important to illustrate the reachable configurations. In the initial configuration d1, the RailCab has the same configuration as the speed control subsystem (cf. Figure 3.1). The unknown vehicle has the configuration of Figure 6.2. The component instances of the types Coordination and MemberCtrl have not been created. Then, either uk:Unknown may create a component instance mc:MemberCtrl and a port for the communication with rc:RailCab by executing the reconfiguration rule createMemberCtrl of Figure 6.4(a) or rc:RailCab may create a component instance co:Coordination executing the reconfiguration rule createCoordinatorCtrl of Figure 6.4(b). This leads to configurations d2 and d3, respectively. Afterwards, the reconfiguration rule of the respective other vehicle may be executed and establish the connection between uk:Unknown and

rc:RailCab. Of course, there are far more configurations reachable in the system indicated by the arrows leaving d4.

For improving the efficiency of the analysis, the reachability analysis uses a depth limited search and identifies isomorphic configurations similar to GROOVE [Ren07]. The depth limitation restricts the length of a path in the labeled transition system to a predefined number of states. Thus, we only investigate the next n configurations that are possible in the system. The identification of isomorphic configurations further reduces the labeled transition system. If we reach a configuration, which is identical up to isomorphism to a configuration that is already contained in the labeled transition system, we identify both configurations and do not investigate the isomorphic configuration a second time.

In our example in Figure 6.10, configuration d4 has been obtained by identifying isomorphic configurations. Applying the reconfiguration rule createCoordinatorCtrl to d2 leads to exactly the same configuration which is obtained by applying createMemberCtrl to d3. If the reachability analysis did not identify isomorphic configurations, the two configurations would be considered as two states, which would lead to a larger labeled transition system.

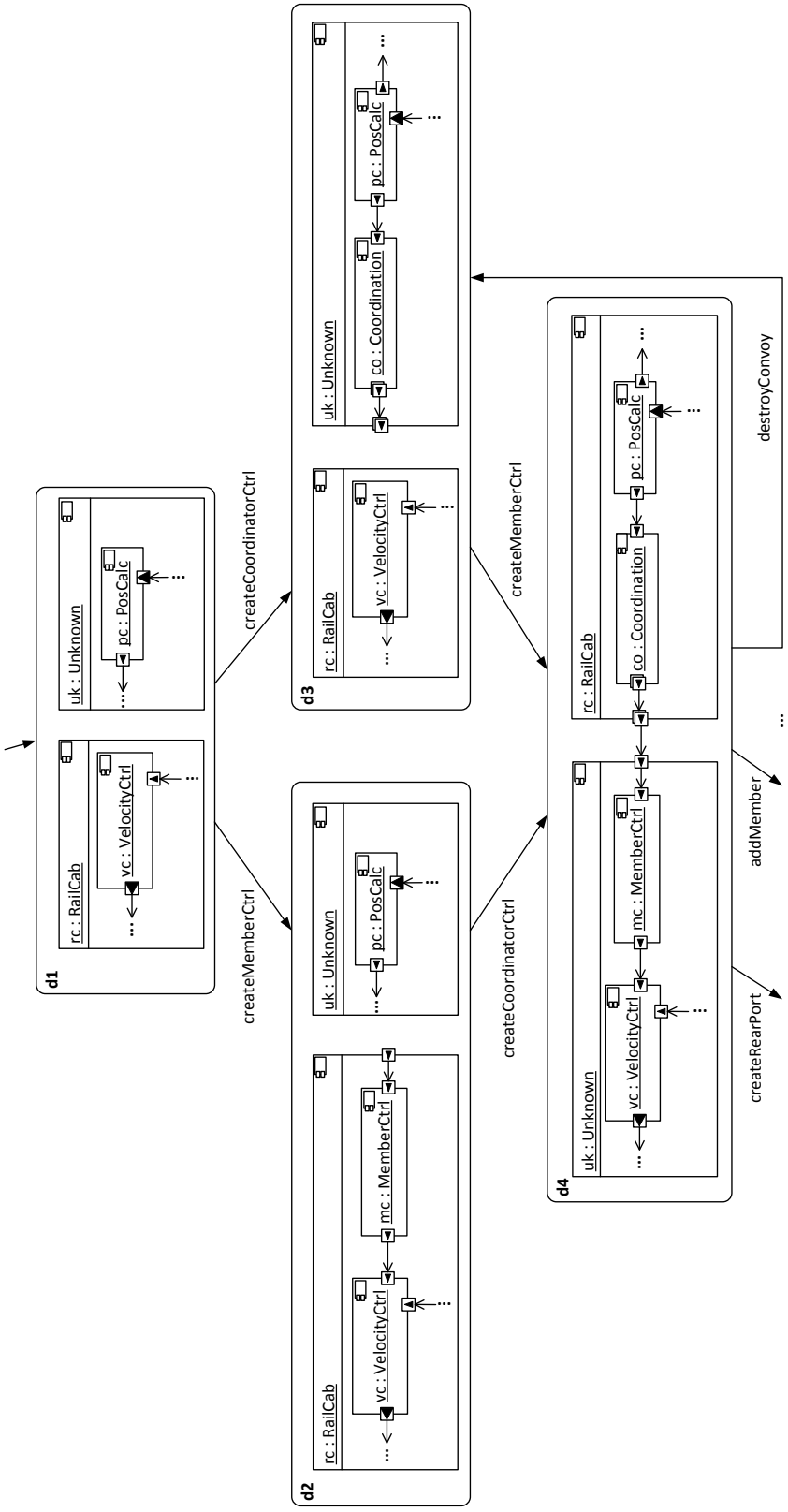


Figure 6.10: Labeled transition system

Figure 6.11 shows the configuration of both vehicles driving in a convoy. It has been constructed by applying the reconfiguration rule of Figure 6.4(a) to the configuration of the unknown vehicle of Figure 6.2 and and the reconfiguration rule of Figure 6.4(b) to the configuration of the RailCab of Figure 3.1.

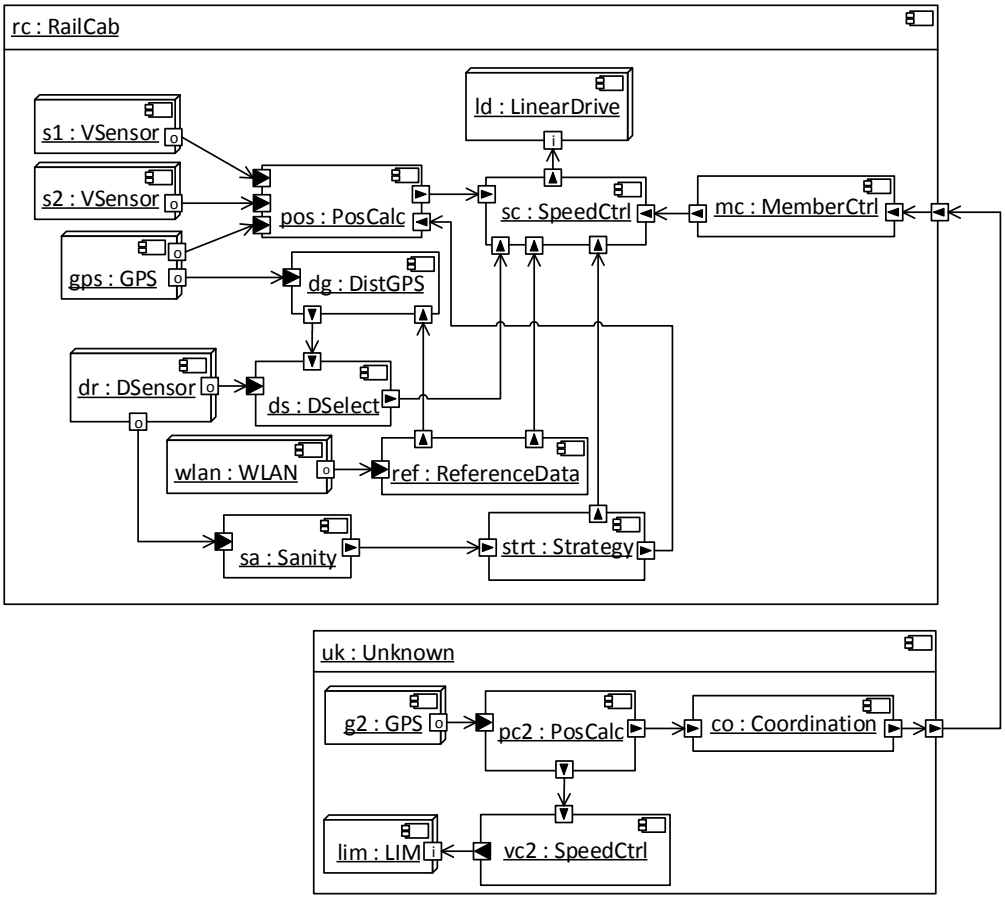


Figure 6.11: Configuration of the two vehicles in convoy mode

Analysis

Figure 6.12 shows the failure propagation model of the two vehicles in convoy mode. The outgoing failure $f^{o}_{sc1.p4,v}$ of the speed controller of the RailCab now also depends on the value error $e_{g2,v}$ in the gps sensor of the foreign vehicle.

This affects the MCS of the hazard *wrong_speed* of the RailCab. The MCS are now $\{e_{dr,v}\}$, $\{e_{s1,v}, e_{s2,v}\}$, $\{e_{gps,v}\}$, $\{e_{wlan,v}\}$, and $\{e_{g2,v}\}$. As presented in Section 5.4, the MCS $\{e_{dr,v}\}$ is successfully removed by the self-healing operation of Figure 3.2. All other MCS including the new MCS $\{e_{dr,v}\}$ remain critical.

The threshold for the occurrence probability of the hazard *wrong_speed* is 0.2 (cf. Section 3.4.3). Without the connection to the foreign vehicle, the self-healing action reduced the occurrence probability of the hazard *wrong_speed* to

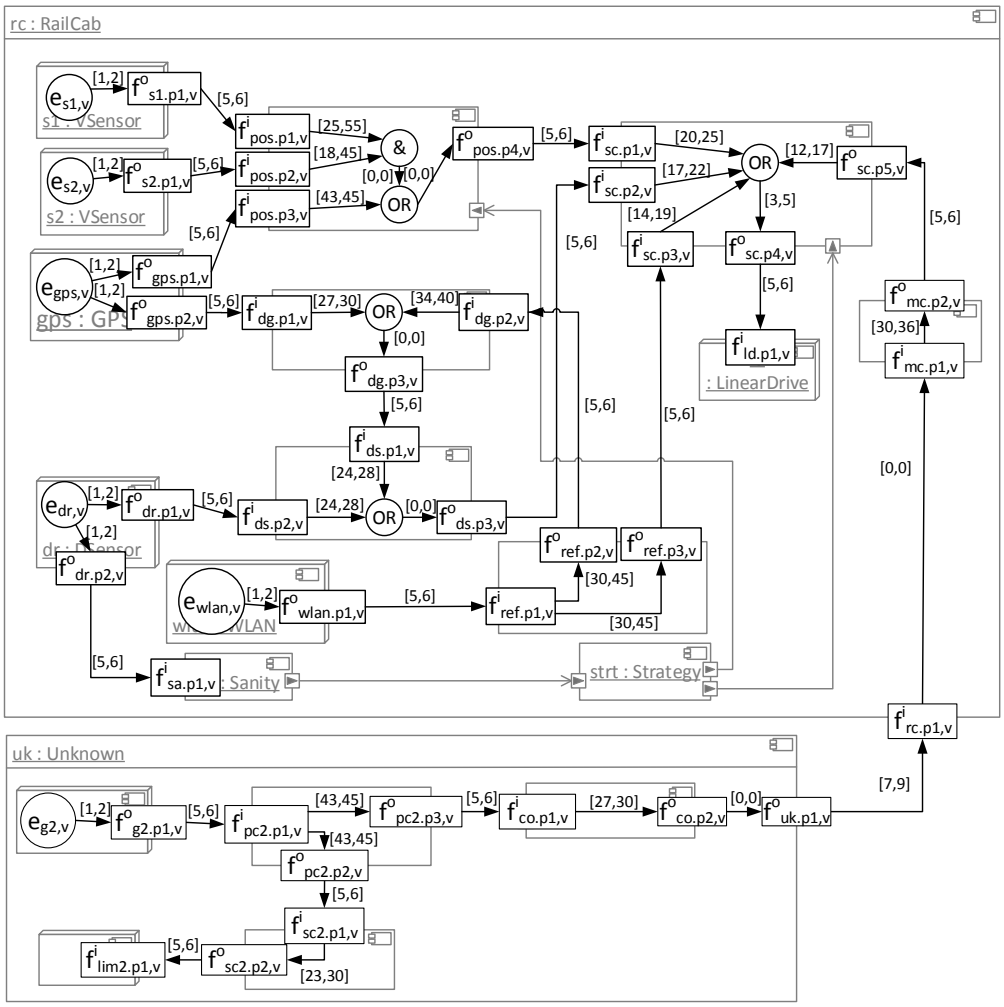


Figure 6.12: Failure propagation model of convoy mode

an acceptable value of 0.1981 (cf. Section 5.5). But, the self-healing operation does not address the value error $e_{g2,v}$ in the GPS sensor of the foreign vehicle. For the case that both vehicles drive in a convoy, the occurrence probability is

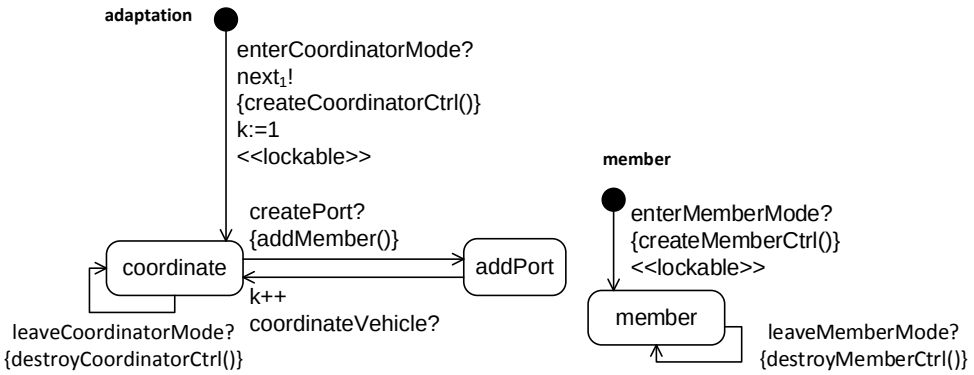
$$\begin{aligned} & 1 - ((1 - p(e_{s1,v}) \cdot p(e_{s2,v}))(1 - p(e_{gps,v}))(1 - p(e_{wlan,v}))(1 - p(e_{g2,v}))) \\ =& 1 - ((1 - 0.01)(1 - 0.1)(1 - 0.1)(1 - 0.1)) \\ =& 0.27829 \end{aligned}$$

The occurrence probability 0.27829 is greater than the threshold of 0.2. The occurrence probability of the hazard *wrong_speed* is thus not acceptable if both vehicles connect to drive in a convoy. As a consequence, the reconfiguration rules of Figure 6.4 need to be locked.

6.2.2 Reconfiguration Controller

In order to stop reconfigurations from being executed, we must lock the transitions that carry these reconfigurations as side effects. Therefore, we introduced *lockable transitions* for real-time statecharts in [PT09]. These transitions are only executed if the safety manager has approved the reconfiguration they are executing as a side effect. Otherwise, the safety manager locks them.

Figure 6.13 shows the real-time statecharts of Figures 6.6 and 6.7 with lockable transitions. In both real-time statecharts, the initial transition is lockable. This is illustrated by the «lockable» label at the initial transitions. The reconfigurations to create coordinator and member control components for convoy communication may be locked because of this attribute. The reconfigurations, which are needed to terminate the convoy, are not lockable.



(a) Adaptation statechart (b) Member role

Figure 6.13: Real-time statecharts with lockable transitions

The initial transitions of the port statechart of the RailCab (cf. Figure 6.13(b)) is locked, because of the risk that was computed above. The initial transition of the adaptation statechart of the unknown vehicle (cf. Figure 6.13(b)) is locked, as well, to keep the system consistent.

Required Reconfigurations

In some cases, locking transitions is not acceptable. Self-healing operations, for example, are required to be executed once a failure has been detected. Otherwise, the safety requirements of the system may be violated. Another example is the synchronization of components. Components that collaborate need to have consistent states. For example, two RailCabs driving in convoy need to provide all communication functionality that is required for convoy drive. Therefore, the subsequent operations must be executed once the vehicles have agreed on building a convoy. This means, generally speaking, that all reconfigurations that are part of a series of operations are required to be executed.

Consequently, not all transitions in a real-time statechart that have a reconfiguration rule attached can be lockable transitions. Therefore, we introduce *required transitions*. Required transitions are not allowed to be locked. They must always be executed once they have been activated.

However, we still need to guarantee that the reconfigurations associated to required transitions are safe with respect to the occurrence probabilities of hazards. Therefore, we partition the paths in real-time statecharts into *lockable partitions*. A lockable partition is a subgraph of the reachable behavior of the real-time statechart with a tree-like structure. It has one start state (root) whose only outgoing transition is the the only lockable transition of the partition. A lock ends at the state or states where the next lockable transitions originate. We analyze all configurations that are part of a lockable partition. If an unsafe configuration is reachable, the whole partition is locked by locking the lockable transition.

Figure 6.14 shows an exemplary path in a real-time statechart. Lockable transitions are represented by bold lines that are labeled with the stereo type «lockable». Transition with required reconfigurations are drawn as thin lines.

We assume that the configuration in State5 is an unsafe configuration and we therefore need to lock the transition from State4 to State5. However, this transition cannot be locked, because the associated reconfiguration is required once we enter State4. The last partition which is lockable on the path from State5 back to the initial state is the transition from State2 to State3. Thus, in order to lock the transition from State4 to State5, we already need to lock the transition from State2 to State3.

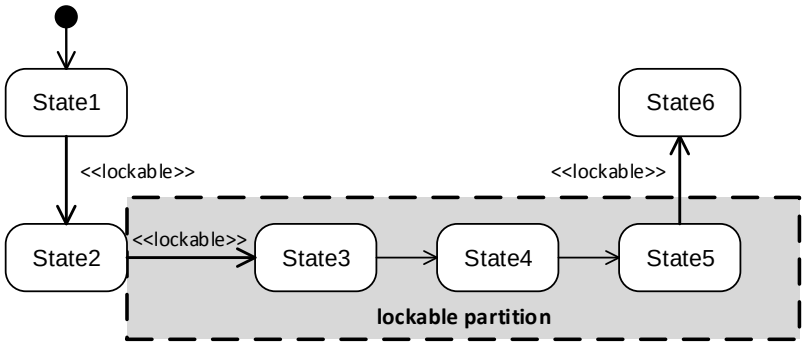


Figure 6.14: Required reconfigurations and lockable transitions

When a required transition is executed, the reconfigured subsystem informs the analyzer about the executed reconfiguration. It is possible that the analyzer processes the update later than the reconfiguration is executed and does not know about the current system state. However, this is no threat to safety as we already excluded unsafe reconfigurations from the last lockable transition preceding the required transitions.

6.3 Risk Analysis

Industrial standards for the development of mechatronic systems consider risks instead of pure hazard probabilities as we have described in Section 2.4.3. The risk is the combination of the occurrence probability of a hazard and the consequences that would result from an accident, namely *severity*, which would be caused by the hazard. This allows for classifying hazards by their criticality [Lev95]. However, the main part of the computation of risk is the computation of the occurrence probability of the hazard. We consequently focus on computing hazard occurrence probabilities for the analysis of self-healing operations at design time.

However, the severity of a hazard may change at runtime. For example, the severity of a collision of two RailCabs may change depending on their load. If both RailCabs are empty, the damage is relatively low compared to the case where one vehicle has loaded dangerous cargo and the other is full of passengers. Consequently, we also consider risk in our runtime analysis as we have published in [PT09, PHST12].

Using the classification of the industrial standard IEC 61508 [Com98] for our example, we assume that the occurrence probability of the hazard wrong speed is classified Occasional (cf. Table 2.1). The severity of a collision of two empty RailCabs is classified Marginal and the collision of a RailCab with dangerous cargo and a RailCab which is full of passengers is classified Catastrophic. In the first case, the risk class is III and I in the second case. If we assume an acceptable risk class of III, the convoy would be permitted for the empty RailCabs. However, the RailCabs with dangerous cargo and the passengers would have to keep long distances between each other.

6.4 Timing Concerns

A timely completion of reconfigurations is of particular importance when executing self-healing operations. If a self-healing operation is completed too late, it may happen that a failure could not be stopped from propagating to the hazard. Consequently, the number of minimal cut sets for the hazard would increase, as would the occurrence probability of the hazard. The increase of the occurrence probability might lead to a too high hazard probability.

To solve this problem, the duration of the runtime analysis must be analyzed. This can for example be achieved by a worst case execution time (WCET) analysis [WEE+08]. A WCET analysis calculates the maximum execution time of a program. The execution time of the runtime analysis does not only depend on the code of the analysis itself but also on the analyzed model. In order to guarantee a WCET, the deployment diagram, reconfiguration rules and failure propagation must be limited in their size and degree of branching.

To improve the execution time of the runtime analysis, of course, configurations that have already been analyzed do not need to be analyzed again. This also holds for configurations that have already been analyzed at design time. The drawback is that more memory is needed to store the results.

With the help of the WCET of the runtime analysis, the developer decides, which reconfiguration are lockable and which are required. For example, for reconfigurations that are executed very quickly in succession, only the first reconfiguration should be lockable. Of course, self-healing operations are always required.

The runtime analysis causes a significant base load. As resources in embedded systems are usually limited, we suggest to execute the analysis on an external computer that is not part of the embedded computing hardware. In large systems (in terms of their physical size) like the RailCab, the analysis may be executed on an additional desktop computer that is integrated in the RailCab. In smaller systems, as for example a miniature robot, the desktop computer may run near the area where the robot operates.

The time needed for runtime analysis does not only depend on the actual analysis described above. In order to analyze the failure propagation between several subsystems, the models of the subsystems need to be exchanged as described in Section 6.2.1. The model exchange is particularly expensive when a subsystem intends to join the system. In this case, the complete deployment diagram and failure propagation models need to be transmitted to the analyzer at once. In this case, the connection of the new subsystem may be delayed. If the new subsystem must connect very quickly, we may assume that the new subsystem is already known and no runtime analysis needs to be performed. We assume that it is improbable that unknown subsystems need to connect to the system urgently.

The transmission time of the models depends on several parameters. Of course, the size of the transmitted data should be as small as possible. Therefore, the transmitted models should be as abstract as possible. They should only contain information that are relevant for hazard analysis. We address this requirement by using TFPGs instead of complete behavior models. TFPGs only contain information of the behavior of errors and failures instead of the whole system behavior. Further, the models must be encoded in an efficient way.

6.5 Summary

In this chapter, we presented a framework that allows for executing AShOp at runtime. This is necessary, because there may be configurations that are constructed at runtime and were unknown at design time. Consequently, the developer could not analyze at design time, which effect self-healing operations have on the failure propagations of these configurations.

Our runtime analysis first computes reachable configurations and then applies AShOp. Configurations, in which hazards occur with unacceptable probabilities, are locked by locking the reconfiguration rules that would result in these configurations. Finally, no such configurations are reachable and the developer can still guarantee that all configuration, which are constructed at runtime, fulfill the safety requirements.

7 Tool Support

The approaches that have been presented in Chapters 4-6 have been implemented as plugins for the Fujaba Real-time Tool Suite [PTH+10]. Fujaba is an Open Source UML CASE tool project, which was kicked off by the software engineering group at the University of Paderborn in 1997. Fujaba was redesigned and became the Fujaba Tool Suite in 2002. The plug-in architecture enables developers to add functionality easily while retaining full control over their contributions. All plugins have been ported to GMF [gmf12] since 2010.

One of the tool suites of Fujaba is the Fujaba Real-time Tool Suite. It supports the modeling and analysis of software in mechatronic systems with MechatronicUML. Therefore, editors for the diagrams of MechatronicUML were implemented. Further, the tool suite provides a tight integration with software tools used by control engineers like MATLAB.

We now give a tour of the plugins that implement the approaches of this thesis. The tour explains the necessary steps that the developer has to perform in order to apply AShOp at design time and at runtime.

7.1 Tour of the Tool

We illustrate the usage of our implementation using the example of the speed control subsystem of the RailCab, which was used in the previous chapters.

7.1.1 AShOp

The application of AShOp requires the following Fujaba diagrams:

- Mandatory
  - An atomic component diagram with the atomic component types
  - A coordination pattern diagram that specifies the coordination patterns
  - A real-time statechart diagram for each atomic component type with a substatechart for each port and the synchronization statechart
  - A component instance diagram which specifies a configuration
  - TFPGs for the hardware nodes in the deployment diagram

  - An error type diagram
  - A failure type diagram
- Optional
  - A structured component diagram with structured component types

We did not implement a deployment diagram editor in Fujaba. This has two reasons. First, hardware nodes are only used to model the occurrence of errors in hardware entities [BBD+12]. This functionality can be implemented for software component instances, as well. Second, during the port of Fujaba to GMF, all editors had to be ported. Since hardware nodes are only used for analyzing hazards, we spent the time implementing other editors like component diagram editors or real-time statechart editors. They are more important for modeling software than a deployment diagram editor. Thus in Fujaba, hardware nodes are represented by component instances and errors are injected into component instances.

TFPG Generation

Before the developer can apply AShOp, he needs to generate TFPGs for the component types. The input model for the TFPG generation are the real-time statecharts of component types (cf. Chapter 4).

We implemented the generation of TFPGs for service and timing failures (cf. Section 4.2.1). The tool for slicing [ACH+12] which is needed to identify value failures (cf. Section 4.2.2) has not yet been integrated.

Our plugin supports the TFPG generation for atomic component types and not for structured component types. TFPGs of structured components are assembled from the TFPGs of their embedded atomic components automatically during later steps of AShOp. The generation of the TFPGs of an atomic component type is executed from the atomic component diagram editor by right-clicking the component type and selecting the item "generate TFPG" from the sub-menu of the item "TFPG generation" (cf. Figure 7.1).

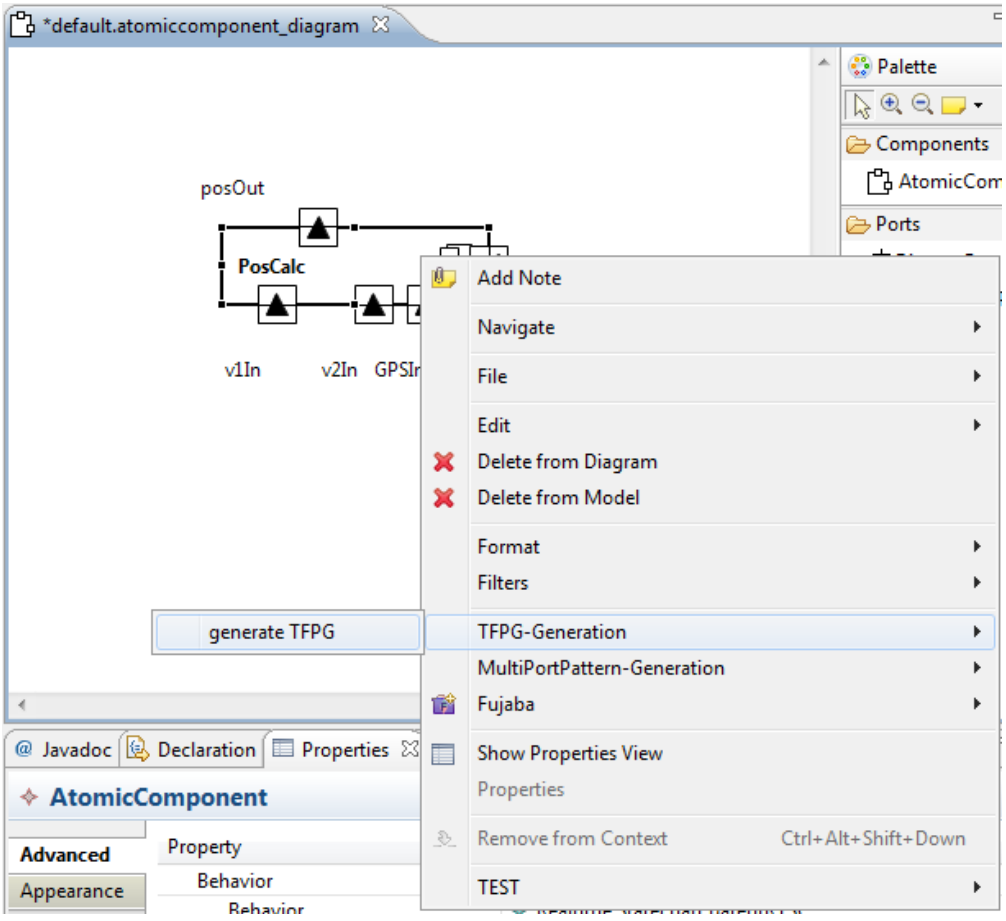


Figure 7.1: Generating TFPGs from real-time statecharts

Figure 7.2 shows the TFPG which has been generated for the outgoing service failure of port posOut of the component type PosCalc. The outgoing service failure is caused by each service or timing failure at the ports v1In, v2In, and modeIn. Failures at the port GPSIn do not affect the outgoing service failure.

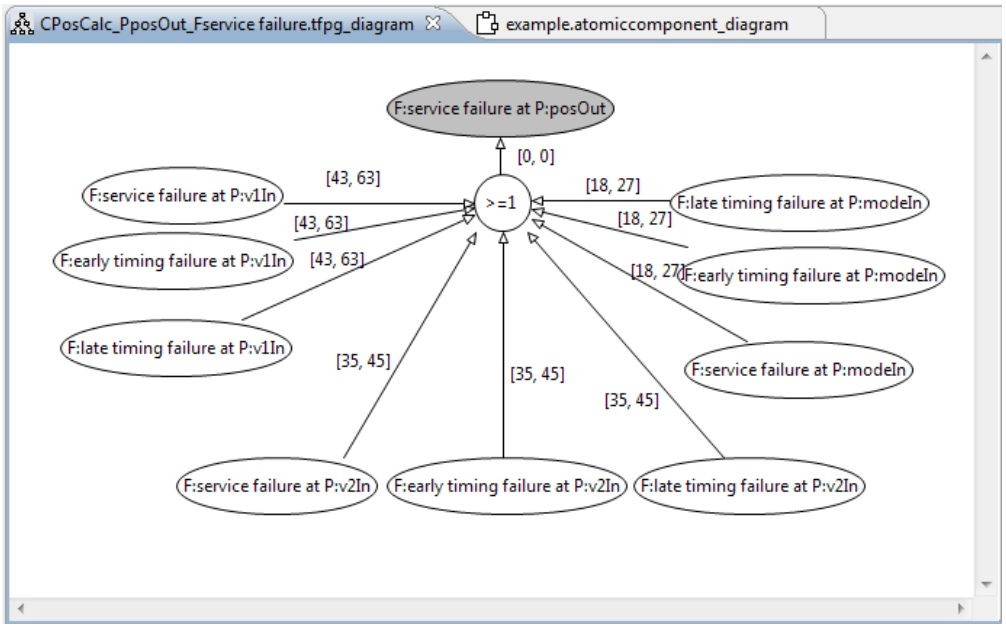


Figure 7.2: TFPG for the outgoing service failure of the component type PosCalc at port posOut

Specifying Hazards and Probabilities

After the TFPGs have been generated, the developer needs to specify occurrence probabilities of errors and define hazards. Therefore, the perspective "Hazard Analysis" has been implemented during the student's project "Automotive Software Engineering" [BBD+07]. This perspective originally provided functionality to specify which errors and failures may occur in components, the failure propagation inside components, the occurrence probabilities of errors in components, and to define hazards. The perspective has been extended in the student's project "SafeBots" [AAB+11] and "SafeBots II" [AGL+12] with functionality to support runtime analyses and AShOp.

Figure 7.3 shows the view "Defect Specification". It is used to specify which errors and failures occur in a component if TFPGs need to be constructed manually. When TFPGs have been generated, the view is used to modify the occurrence probabilities of errors. Figure 7.3 shows the errors of the component GPS. The errors, which have been generated for this component, are displayed in the text box on the right. Initially, the occurrence probabilities of the generated errors are set to zero. This value is modified by selecting an error in the right text box, entering the new value in the text box in the center, and then pressing the button "modify". In Figure 7.3, the occurrence probability of the service error has already been set to 0.01. The early timing error is selected and will also be set to 0.01.

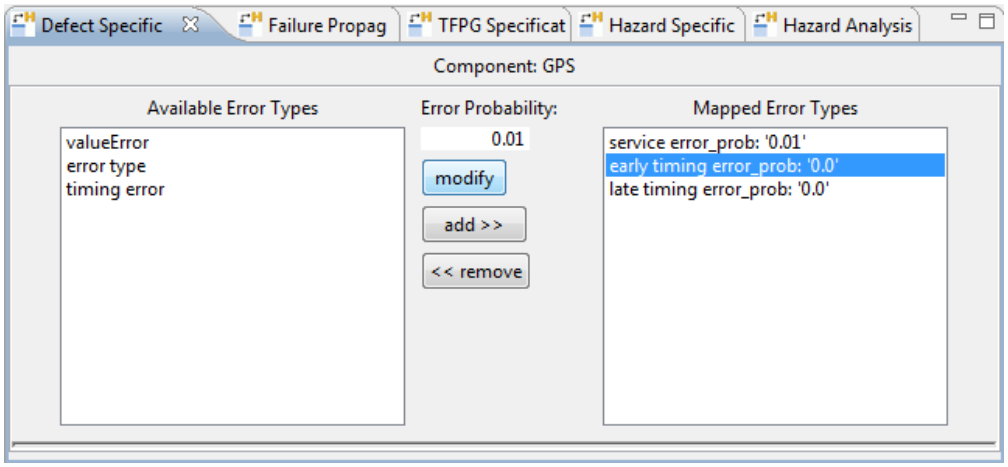


Figure 7.3: Specifying error probabilities

The last action before executing AShOp is the specification of the hazard. This is done in the "Hazard Specification" view as shown in Figure 7.4. First, a component instance diagram is opened. To activate the editing of the hazard for the component instance diagram, the user clicks into a white space in the component instance diagram.

The left text box of the "Hazard Specification" view shows the specified hazards, the center text box is used to enter the Boolean formula, and the right text box

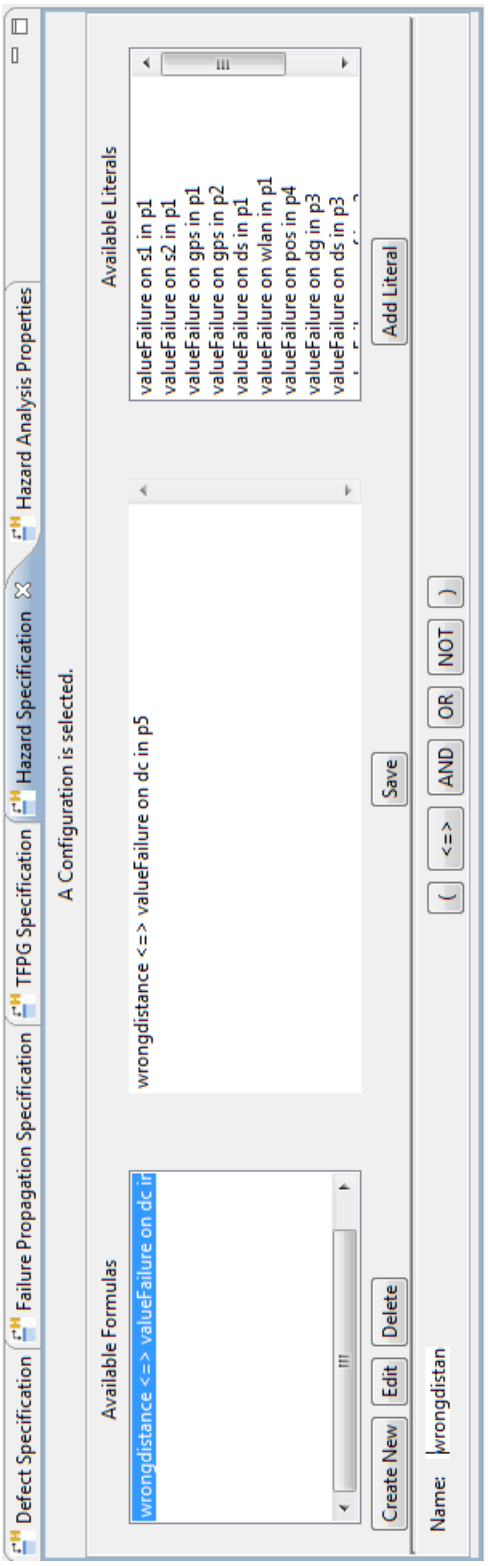


Figure 7.4: Specifying hazards

shows which failures are specified for the component instances in the component instance diagram. To specify a hazard, the name of the hazard is entered in the small text box on the left bottom. Then, the button “Create New” above this text box is pressed. The name of the hazard will appear in the text box in the center. The Boolean formula, which specifies the hazard, is then entered in the center text box using either the operator buttons at the bottom and the failures in the right text box or by typing the formula. The formula is saved by pressing the “Save” button below the center text box. Afterwards, the formula will appear in the left text box. The formula may be modified by pressing the button “Edit” or deleted by pressing the button “Delete” below the left text box.

Manual Specification of TFPGs

In some cases, TFPGs cannot be generated. Hardware nodes, for example, do not have a state-based behavior by definition (cf. Section 2.2.1). In order to specify a failure propagation for these hardware nodes, the developer may add a state-based behavior or model the TFPG directly. Since hardware nodes are modeled by software component in Fujaba, TFPGs can be generated. However, if no behavior is specified for hardware nodes, it is easier to simply specify a TFPG because hardware nodes often have a simple failure propagation. Mostly, errors are simply connected to failures without logical operators.

Before the developer can model a TFPG, the classes of errors and failures (cf. Section 2.4.2) which may occur in the system and their hierarchy must be specified. Therefore, the error type diagram editor and the failure type diagram editor are used. Figure 7.5 shows the failure type diagram editor with a failure type hierarchy. This editor has first been implemented in the course of the project group ASE [BBD+07] and has been implemented in GMF during the port to GMF in 2010.

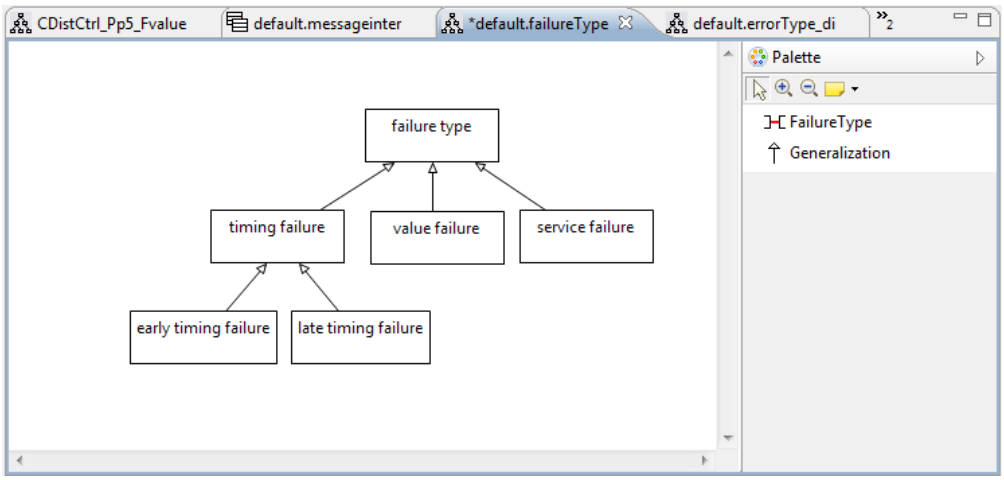


Figure 7.5: Specifying failure types

Afterwards, the error and failure types are added to component types and port types. This is done in the atomic component editor using the view "Defect Specification". Figure 7.6 shows an atomic component diagram editor and the "Defect Specification" view. A failure type is added to a port by selecting the port in the atomic component editor and adding a failure type in the Defect Specification view of the Hazard Analysis view. The example in Figure 7.6 shows the atomic component type GPS. Port Out1 is selected and the failure type early timing failure is added to this port. Occurrence probabilities of errors are specified as described in the text of Figure 7.3.

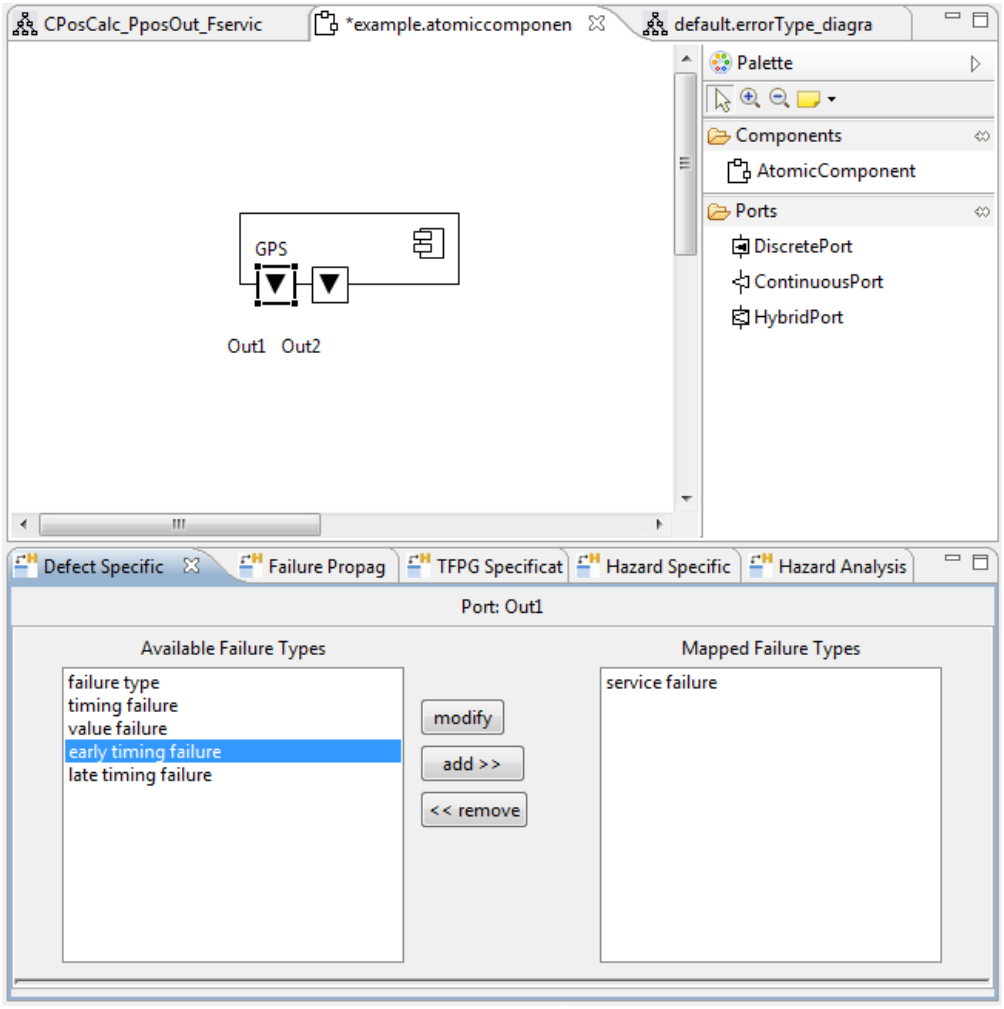


Figure 7.6: Specifying failures at ports

When all error and failure types have been added to a component type, the TFPGs of this component type can be modeled in the TFPG editor. For this, the view "TFPG Specification" of the Hazard Analysis perspective is opened and a port with outgoing failures is selected as shown in Figure 7.7. Then the outgoing failure that will become the root node of the new TFPG is selected in the drop down menu in the "TFPG Specification" view. In Figure 7.7, value failure is chosen. To create the TFPG, the button "Specify TFPG" is pressed and the TFPG editor opens.

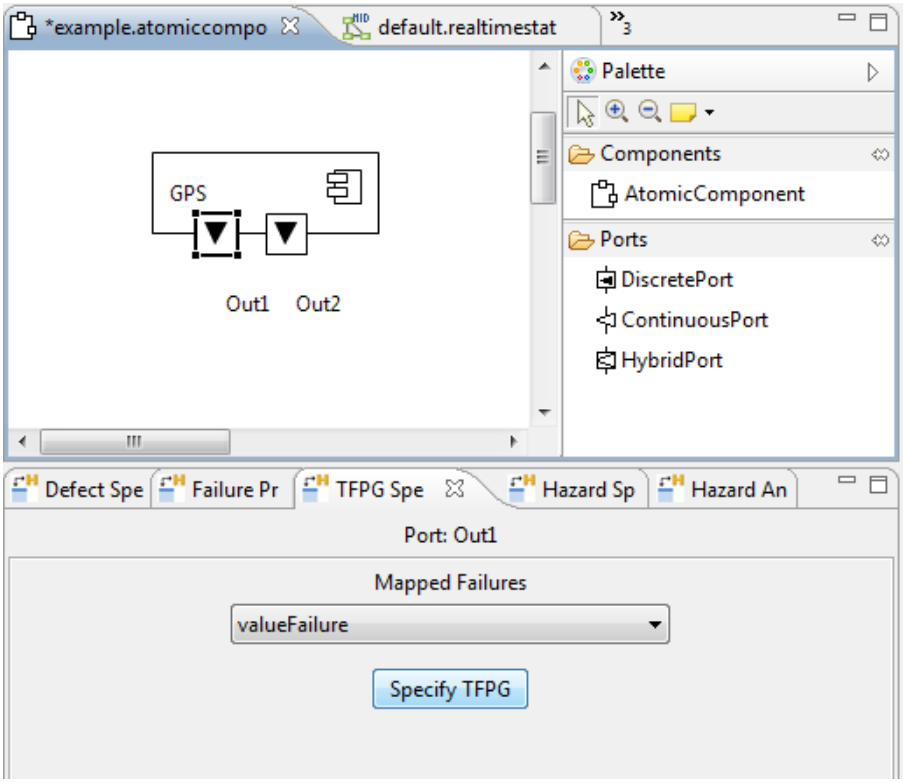


Figure 7.7: Creating a TFPG for port type p3 of component type DistGPS

Figure 7.8 shows the TFPG editor with the TFPG of an outgoing value failure at port Out1 of the component type DistGPS. This editor has been implemented in the course a of the student's project "SafeBots II" [AGL+12].

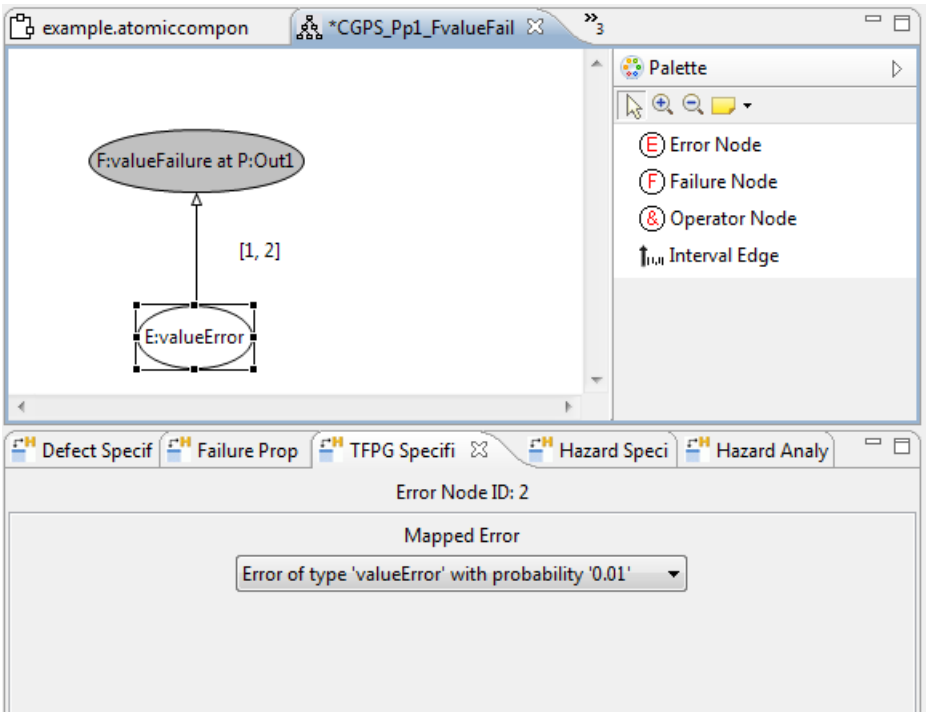


Figure 7.8: Modeling the incoming failures of a TFPG

When the editor is opened, only the root node is present. It is filled with gray color. In Figure 7.8, the root node is the outgoing value failure of port Out1. It is created from the failure of the port which was selected in the atomic component editor.

The TFPG editor uses the palette at the right and the view "TFPG Specification" to edit the created TFPG. Nodes which represent errors and incoming failures are created using the tools "Error Node" and "Failure Node" from the palette on the right. Error and failure nodes are associated to concrete errors and failures using the drop down menu in the "TFPG Specification" view. In our example of Figure 7.8, the value error of the component is selected.

Operator nodes are created by the tool "Operator Node" from the palette. Initially, the operator is set to &. The value may by changed to ≥ 1 in the properties of the operator node.

Edges are created by the "Interval Edge" tool. The source node is selected an the edge is drawn by dragging the edge to its target node. Initially, the propagation time interval of the edge is set to $[0, 0]$. It is modified by selecting the interval and entering the new values.

AShOp

AShOp may be applied when the TFPGs have been created, the occurrence probabilities of the errors have been specified, and the hazards have been defined. This part of the tool has been implemented in the course of the student's project "SafeBots II" [AGL+12].

AShOp is started from the component instance diagram editor. Figure 7.9 shows the component instance diagram editor with a component instance diagram that represents the configuration of the speed control subsystem (cf. Figure 3.1) that has been use throughout this thesis. The user right-clicks on a white space in the diagram and a menu pops up. The sub-menu "Analyze Self-healing Operation" under menu item "HazardAnalysis" will open the window of Figure 7.10 from which the analysis is triggered.

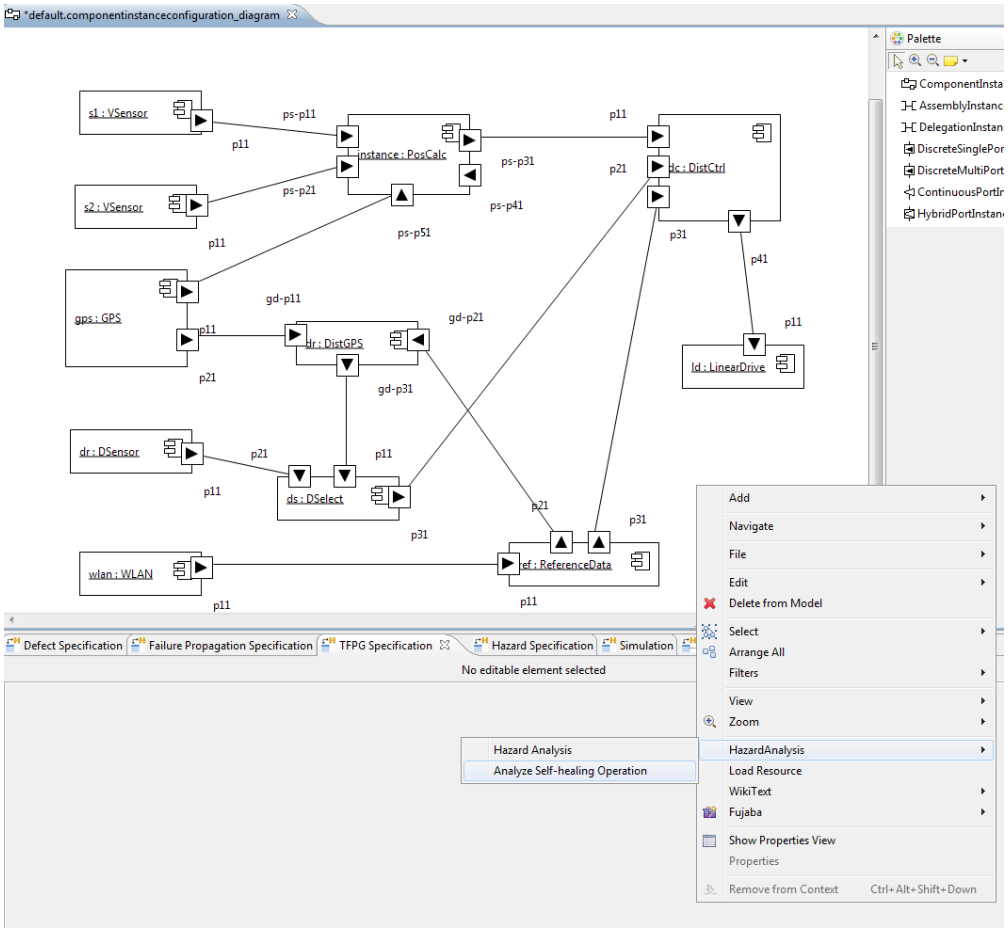


Figure 7.9: Executing AShOp

Figure 7.10 shows the window from which AShOp is triggered. Initially, the text box in the center is empty. AShOp is triggered by pressing one of the two lower buttons. If "Analyze" is pressed, the analysis will operate on the affected part as described in Section 5.3. This is the common use case. If "Analyze full graph" is pressed, the analysis will use the whole configuration. This button has been implemented for evaluation (cf. Section 7.3.3).

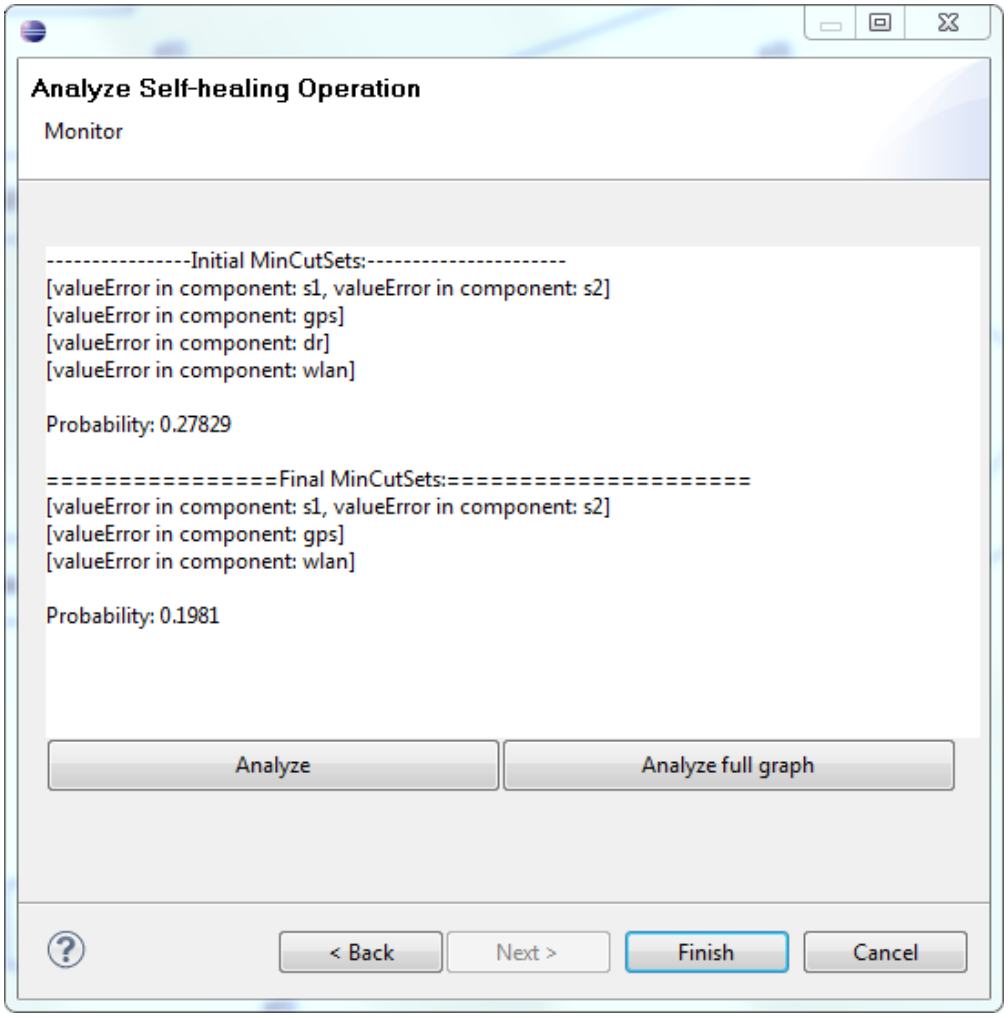


Figure 7.10: Result window

When one of the buttons has been pressed, a dialog opens where the self-healing operation is selected. Then, the analysis starts.

When the analysis is finished, the window lists the minimal cut sets before and after the application of the self-healing operation and the according hazard occurrence probabilities. Initially, the hazard wrong_speed has the minimal cut sets $\{e_{s1,v}, e_{s2,v}\}$, $\{e_{gps,v}\}$, $\{e_{dr,v}\}$, and $\{e_{wlan}\}$. After the self-healing, the MCS $\{e_{dr,v}\}$ has been removed.

Below the minimal cut sets, the occurrence probability of the hazard is displayed. Before the application of the self-healing operation, the probability of the hazard is 0.27829. After the self-healing operation, the probability of the hazard has been reduced to 0.1981.

7.1.2 Runtime Analysis

We implemented a runtime analysis that prevents unsafe configurations as described in Chapter 6. It computes the hazard occurrence probabilities of all reachable configurations and locks reconfigurations that exceed the acceptable hazard occurrence probabilities. However, this analysis does not yet take self-healing operations into account. Still, the same framework that computes reachable configurations (cf. Section 6.2.1) and controls reconfigurations (cf. Section 6.2.2) may be used to analyze self-healing operations at runtime. Only the analyzer (cf. Section 6.2.1) needs to be extended by the functionality to execute AShOp. This part of our implementation has been implemented in two student's projects [Nig09, AAB+11].

The runtime analysis needs to perform three different tasks:

1. Compute the reachable configurations
2. Analyze the reachable configurations
3. Lock unsafe configurations

The reachability analysis has already been implemented as a plugin for the Fujaba Real-time Tool Suite. The reachability analysis and the hazard analysis of Giese et al. [GT06] have been integrated into a framework during the student's project SafeBots [AAB+11] to allow for the execution at runtime.

The configurations, which are analyzed by the runtime analysis, are created at runtime. This means, hazards cannot be specified for configurations any more. Therefore, the hazard specification has been altered such that hazards may be specified for component types [AAB+11]. Figure 7.11 shows the view "Hazard Type Specification" for the specification of hazards associated to failures of component types. It has been adapted from the "Hazard Specification" view. When a component type has been selected, the outgoing failures of the component's port types become visible in the right text box. First, the name of the hazard is specified in the text box labeled with "Name:". By clicking the "⇔" button, the hazard and the equivalence operator become visible in the center text box. In the example in Figure 7.11 this is "wrong_speed ⇔". Then, failures from the right text box may be added by selecting them and clicking "Add Literal". Operators and brackets are added by clicking the buttons below the center text box. The formula is saved by clicking the "Save" button below the center text box.

In order to lock reconfigurations at runtime, transitions that are labeled with reconfigurations must be lockable (cf. Section 6.2.2). The lockable attribute

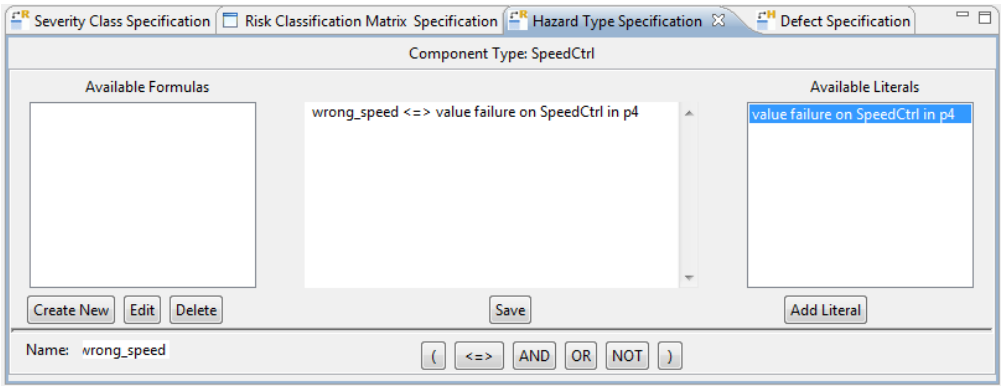


Figure 7.11: Specifying hazard types

of a transition is set in the "Properties" view as shown in Figure 7.12. The transition is selected and the attribute "Lockable" is set to true. Initially, the attribute is set to false which means that the transition is required.

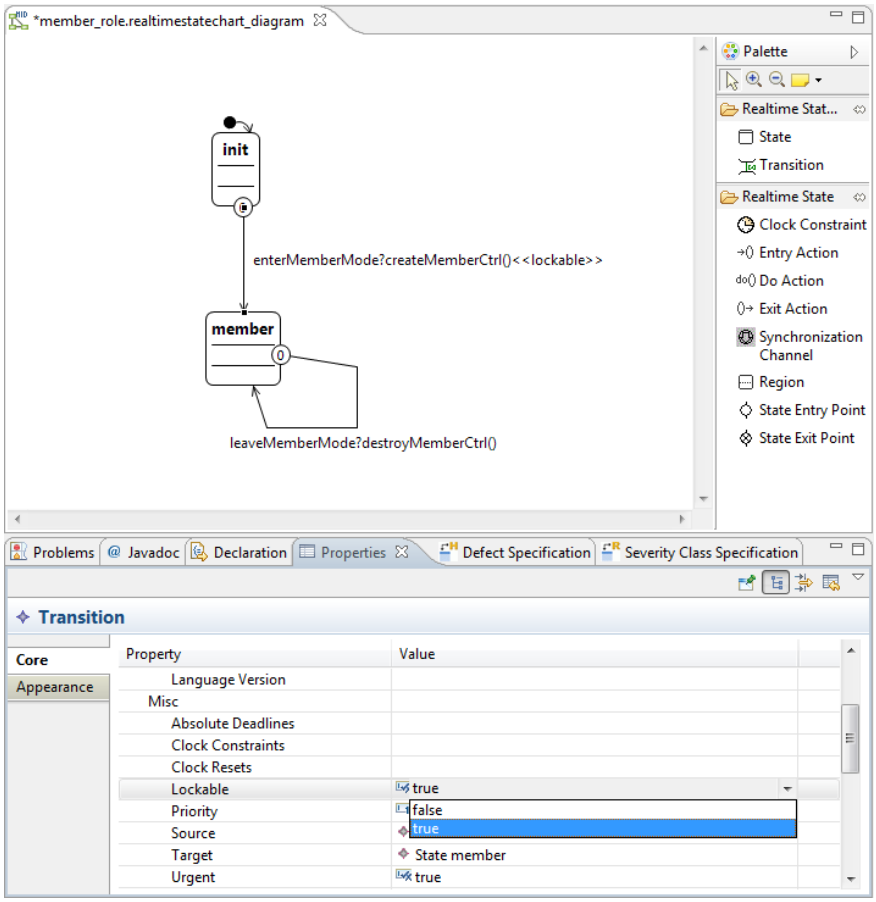


Figure 7.12: Specifying lockable transitions

Risk Analysis

In order to analyze the risk, we need to provide functionality to specify the required parameters, like the severity of a hazard (cf. Section 2.4.3). The risk analysis plugin provides a user interface for the specification of these parameters. The user interface has been implemented in the "Risk Analysis" perspective. The parameters, which are specified using the risk analysis perspective, are the frequency classes, the severity classes, the risk matrix, hazards for component types, and severities for hazards. In the following, we present the views of the risk analysis perspective. All views of this perspective are used with the atomic component diagram editor. In order to edit the widgets of the views, a component type needs to be selected in the atomic component diagram editor.

Figure 7.13 shows the view "Frequency Class" for the specification of frequency classes. The name of the frequency class is specified in the leftmost text box. The next two text boxes take the lower and upper bounds of the frequency class. The "Create" button saves the frequency class which then becomes visible in the rightmost text box.

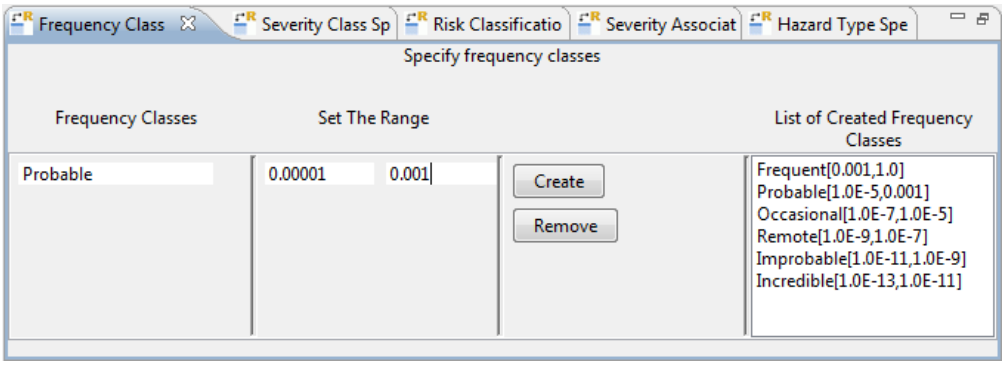


Figure 7.13: Specifying frequency classes

Figure 7.14 shows the view "Severity Class" for the specification of the severity classes. The leftmost text box takes the name of the severity class. The integer values of the severity class are specified in the next two text boxes. Pressing the "Create" button saves the severity class in the Fujaba model file. Then, the severity class becomes visible in the rightmost text box.

Figure 7.15 shows the view "Risk Classification" with which the relations between the frequency and severity classes may be specified in a risk classification matrix. Here, a standard may be loaded from an XML file. The syntax of the XML specification has been defined in [AAB+11]. The risk classes may also be added manually by filling out the cells in the table.

Figure 7.16 shows the "Severity Association Specification" view for the association of hazards and severity classes. The hazard is selected in the leftmost text box. Then, a severity class is chosen from the drop-down menu. Pressing the

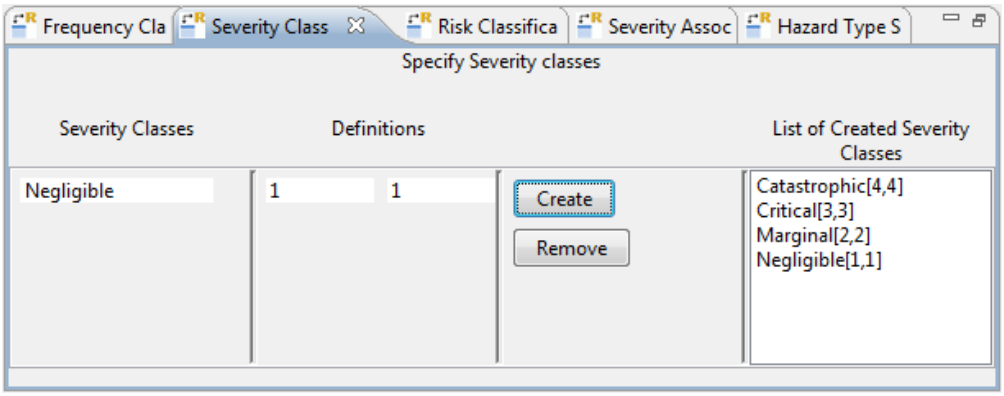


Figure 7.14: Specifying severity classes

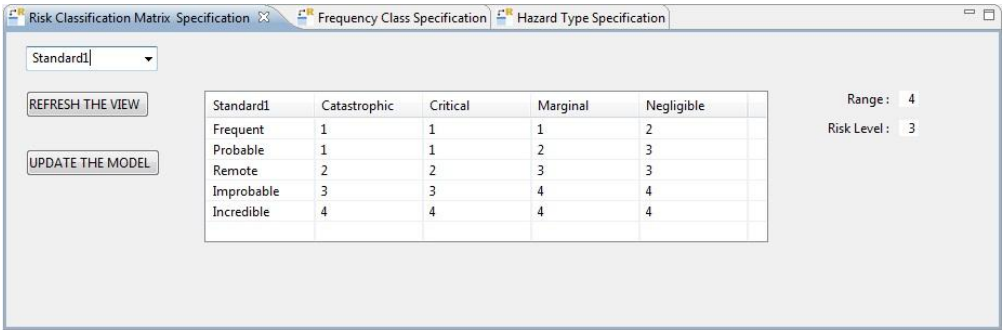


Figure 7.15: Specifying a risk matrix

"Create" button saves the association in the Fujaba model file. In the example of Figure 7.16 the hazard wrongSpeed is selected. It is associated with the severity "Marginal", because initially there are no passengers in the RailCab.

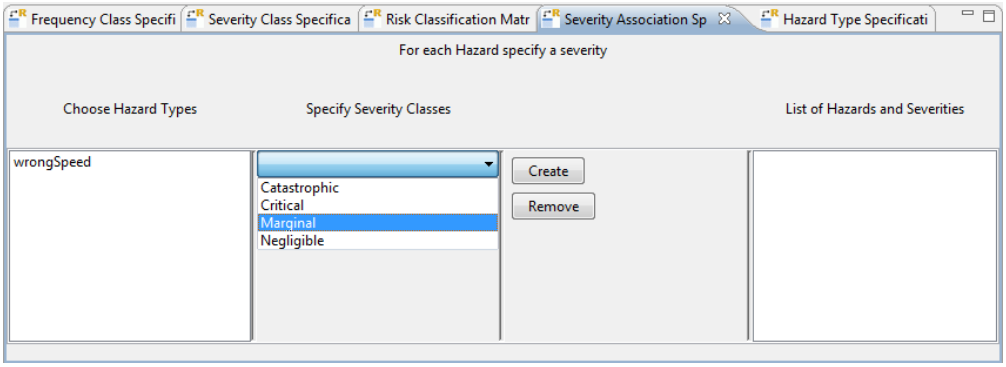


Figure 7.16: Specifying hazard severities

7.1.3 Simulation

The runtime analysis was executed in a simulation and on miniature robots in the course of the student's project "SafeBots" [AAB+11]. For this, we used Player/Stage [VGc12]. Player/Stage consists of the two parts *Player* and *Stage*. Stage is a simulation platform. Player is a network server to control robots. It acts as a hardware abstraction layer for robots by providing interfaces to the robot's sensors and actuators via the IP network. The Player library provides a set of interfaces from software to robotic hardware. Code which is linked against the Player library may be used in the simulation with Stage and (without modification) on a robot which supports Player.

Figure 7.18 shows an example of a simulation in Stage. There are two vehicles that drive in a convoy. The vehicles are represented by gray squares. The red triangles in the square indicate the head of the vehicle. The bold lines represent solid walls in the environment.

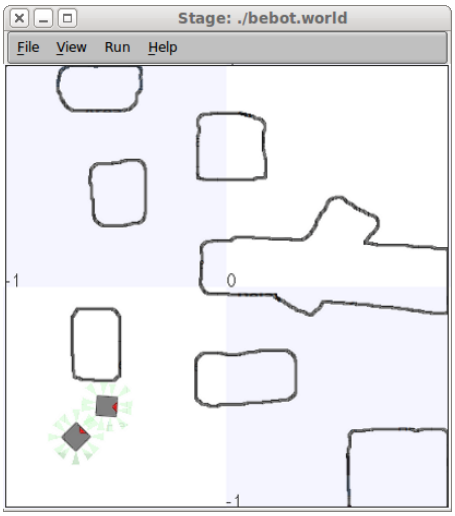


Figure 7.17: Two BeBots driving in a convoy in Stage

A code generation that generates C++ source code from MechatronicUML models was implemented during the student's project "SafeBots I" [AAB+11]. The source code in particular uses the Player library.

The state of each vehicle in the simulation is displayed in the "Status View" view. It synchronizes with the executed system via the Player server. Figure 7.18 shows the status view of the simulation of Figure 7.17. Each line in the status view displays the state of one subsystem. In this example, each vehicle is a subsystem. The first column shows the ID of the subsystem. The second column displays the active state in the real-time statechart. The third column shows which reconfiguration rules have been locked. In the example, the subsystem localhost:6665 is in state Initial and the reconfiguration rule goMember() has been locked.

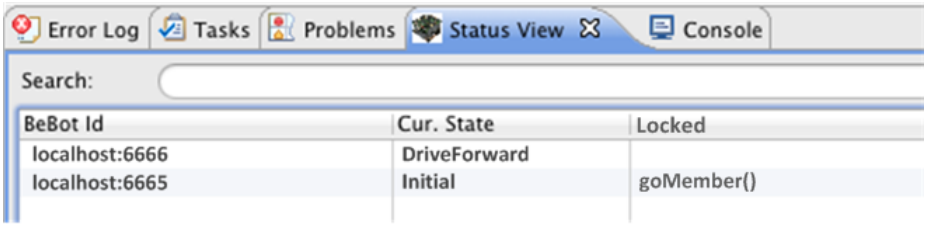


Figure 7.18: Status window of the simulation

7.2 Software Architecture

Figure 7.19 shows the architecture of our plugin. Plugins that already existed before the implementation of AShOp are Core, SDMMetamodel, StoryDiagramInterpreter, SDMReachabilityAnalysis, StoryChecking, MechatronicUMLMetamodel, and HazardAnalysis. These plugins are shown in gray. Plugins that have been implemented in the course of this thesis are THA, HazardAnalysisMetamodel, TFPGMetamodel, TFPGGeneration, RiskAnalysis, and RiskAnalysisMetamodel. These plugins are shown in black.

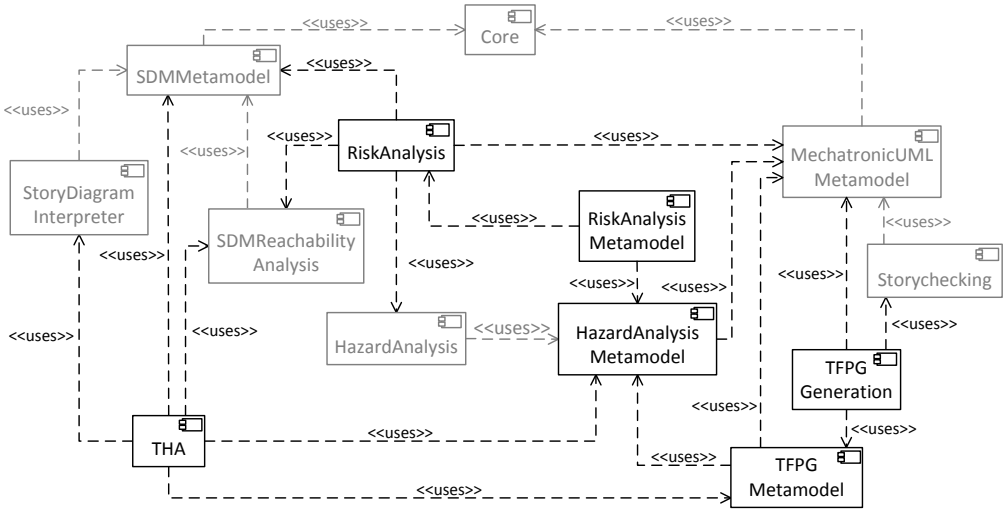


Figure 7.19: Architecture of the AShOp plugin

Core contains the core functionality, e.g., the extension mechanism that allows to integrate plugins. The MechatronicUMLMetamodel plugin contains all classes needed to construct MechatronicUML-models. The SDMMetamodel plugin provides the classes to construct story diagrams. The HazardAnalysisMetamodel plugin contains the elements for modeling failure propagation formulas of the HazardAnalysis plugin which have been redesigned for GMF. The failure propagation formulas are now implemented conforming with expressions from the SDMMetamodel plugin. The TFPGMetamodel provides the data structure for storing TFPGs (cf. Section 3.4).

Figure 7.20 shows the TFPG meta-model. The main class is the class TFPG. The meta-model specifies TFPGs according to Def. 3.4.2. A TFPG has an arbitrary number of Node and Edge objects. Node objects may be objects of the subclasses ErrorNode, FailureNode, or OperatorNode. The attribute operation of the class OperatorNode may either be AND or OR. Objects of the type Edge are always of the subtype IntervalEdge. The attributes lowerBound and upperBound of IntervalEdge specify the lower and upper bound of a propagation time interval.

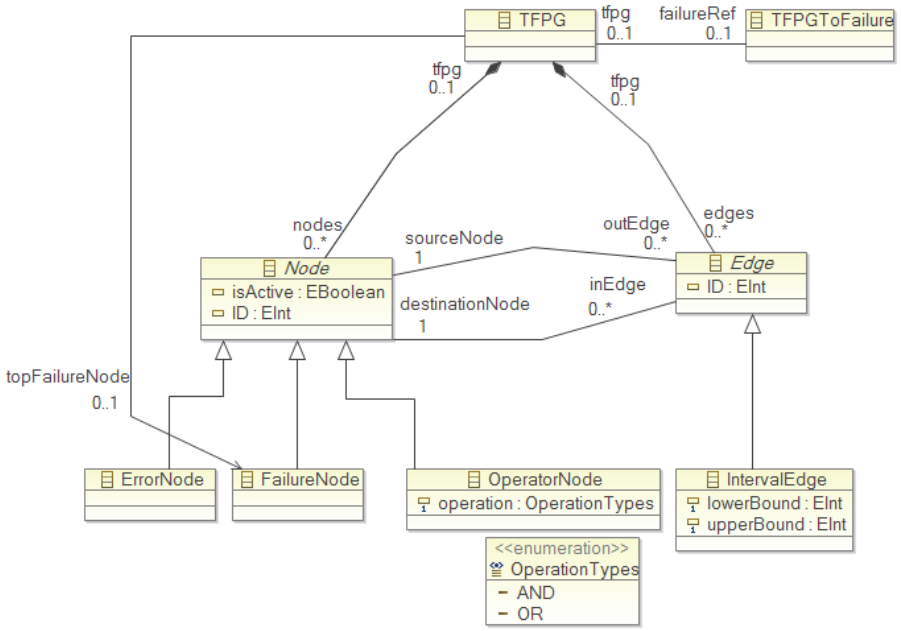


Figure 7.20: TFPG meta-model [AGL+12]

Generation of TFPGs

The TFPGGeneration plugin is used to generate TFPGs from real-time statecharts. It uses the MechatronicUML metamodel plugin to parse real-time statecharts. This plugin provides the data structures for real-time statecharts which are the source models of the generation of TFPGs (cf. Chapter 4). The Storychecking plugin is used to compute the reachable behavior of component types (cf. Section 4.2.1). The HazardAnalysis meta-model is used to create error and failure variables.

AShOp

The main algorithm of AShOp is implemented in the THA plugin. The acronym THA stems from the initial name of AShOp which was "timed hazard analysis". The THA plugin uses the TFPGMetamodel plugin to store TFPGs. The HazardAnalysis plugin implements the hazard analysis of Giese et al. [GT06]. It is used

to compute the minimal cut sets and hazard occurrence probabilities which are needed to analyze the criticality of MCS after the application of the self-healing operation (cf. Section 5.4) and the success of the self-healing operation (cf. Section 5.5). The SDMReachabilityAnalysis plugin provides a reachability analysis on component instance configurations and story diagrams of Heinzemann et al. [HSE10]. This reachability analysis is used to determine the reconfiguration delay.

We use the external tool "Roméo" [LRST09] to compute the locations of errors and failures (cf. Section 5.3). "Roméo" is integrated in the THA plugin by adapters and provides a reachability analysis for TPNs. The transformation from TFPG to TPN is also implemented in the THA plugin.

The StoryDiagramInterpreter plugin evaluates story diagrams on component instance configurations. It is used to apply the self-healing operation to the analyzed configuration (cf. Section 5.3). We use an altered version of the StoryDiagramInterpreter plugin to determine the matching of the story diagram and compute the affected subgraph (cf. Section 5.3).

Runtime Analysis

The main plugin of the runtime analysis is the RiskAnalysis plugin. It executes the risk analysis at runtime. It provides the algorithms for the evaluation of the reachable component instance configurations of the system, for the computation of the risk class, and for the decision about allowing or locking a reconfiguration. The SDMReachabilityAnalysis plugin is used to compute the reachable configurations. The HazardAnalysis plugin is used to compute hazard occurrence probabilities of configurations. The hazard occurrence probabilities are used by the RiskAnalysis plugin to compute the risk. The SDMMetamodel and MechatronicUMLMetamodel plugins are needed to store real-time statecharts and story diagrams.

The RiskAnalysisMetamodel plugin provides data structures to specify the parameters specific to risk analysis, e.g., severity class or risk classification matrix. It uses the HazardAnalysisMetamodel plugin to associate hazards with severity classes.

7.3 Evaluation

We evaluated our implementation of AShOp at design time by conducting three case studies. First, we applied AShOp to the RailCab using the example of the speed control subsystem, which we introduced in Section sec:tha-example. This case study is subject of Section 7.3.1. The other two case studies analyzed the scalability of the generation of TFPGs (cf. Section 7.3.2) and the analysis of self-healing operations (cf. Section 7.3.3) using toy examples. The goal of these two case studies was to find a limit in the size of the models to which

the approaches are applicable and the evaluation of the execution time of the TFPG generation and AShOp with respect to the size of the model.

All case studies were executed on a SuSE 11.4 machine with 72 GB RAM and 8 64-bit CPUs with 2.2 GHz clock and 8 MB cache memory. The implementation uses only one core of the CPU.

7.3.1 RailCab

We applied AShOp to the speed control subsystem that we introduced in Figure 3.1 and the self-healing operation of Figure 3.2. The execution took 20 seconds.

The TFPGs of the hardware nodes VSensor, GPS, DSensor, WLAN, and LinearDrive have been constructed manually. The TFPGs of the component types PosCalc, DistGPS, DSelect, ReferenceData, and DistCtrl have been generated. The generation of the TFPG for the component type DSelect was the fastest and took 0.25 seconds.

7.3.2 Identification of Relations Between Incoming and Outgoing Timing and Service Failures

The identification of relations between incoming and outgoing timing and service failures is conducted separately for each path in the zone graph that represents the reachable behavior of a component type (cf. Section 4.2.1). We consequently evaluate the scalability of this identification for a real-time statechart that consists of a single path only. For the generation of TFPGs from multiple reachable paths in a real-time statechart, the execution times of the TFPG generation for the different pathes are summed up.

The input for the experiment is a real-time statechart that consists of one path only. The last transition has an output message (output transition). All other transitions have input messages (input transitions). The time constraints only need to fulfill the requirement that no deadlock is created. The experiment starts with a path that has one input transition and one output transition. For each step of the evaluation, one input transition is added to the path. We omit silent transition from the input model, because they do not affect the reachability analysis significantly. The reason is that no context automata are generated for silent transitions, which would significantly enlarge the reachable behavior.

Figure 7.21 shows the result of our experiment. The curve indicates that the TFPG generation is of exponential complexity. The largest model, which could be used, was a path with seven input transitions. A path with eight input transitions resulted in a memory overflow.

During the evaluation, we noticed that the reachability analysis is the most important factor of the TFPG generation. 95% of the generation is spent for

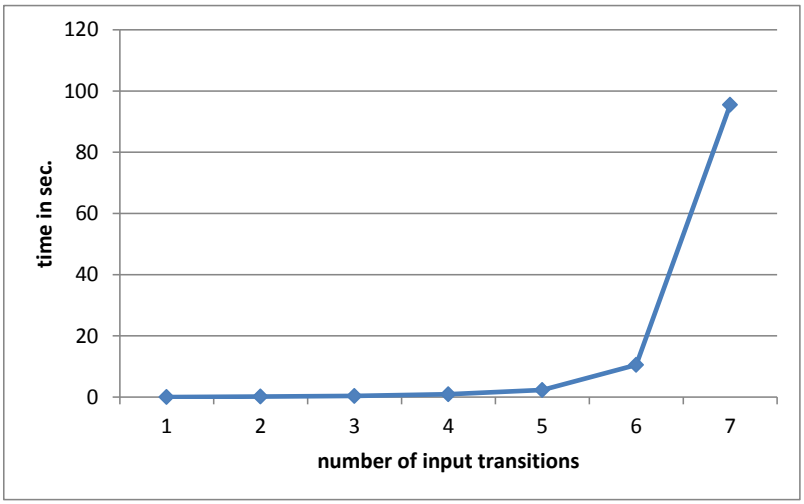


Figure 7.21: Runtime of TFPG generation depending on the number of incoming messages

the computation of the reachable behavior. Only 5% is spent for generating contexts, evaluating the reachable behaviors, computing propagation times, and constructing TFPGs. We consequently propose to improve the runtime of the reachability analysis to improve the runtime of the TFPG generation.

7.3.3 AShOp

We evaluated AShOp using two toy example TFPGs with different structures. One TFPG has a nearly linear structure while the other is a binary tree. With these two examples, we aimed to investigate how the branching degree of the TFPG affects the applicability and the runtime of the analysis. This was achieved by applying AShOp repeatedly to models of increasing sizes.

Further, we evaluated how the execution time is affected by the reduction of the reachability analysis to the part of the TFPG which is affected by the self-healing operation as described in Section 5.3. We therefore applied AShOp to the same TFPG twice: Once using the affected part and once using the full graph.

We omitted the computation of the critical time, which has been described in Section 5.2.

Linear Structure

Our linear TFPGs have a structure as illustrated in Figure 7.22. The TFPG consists of the two paths $e_1, \ldots, f_{j3}$ and $e_2, \ldots, f_{j3}$. The reconfiguration deletes failure variables f_n and f_{j1} and the edges between them (highlighted in red). Thus, the affected part is built by the path $e_1, \ldots, f_{j1}$. For evaluation, we

extended the length of both paths equally by one component instance per evaluation step. We repeated the experiment with different numbers of paths in the affected part as sketched in Figure 7.22.

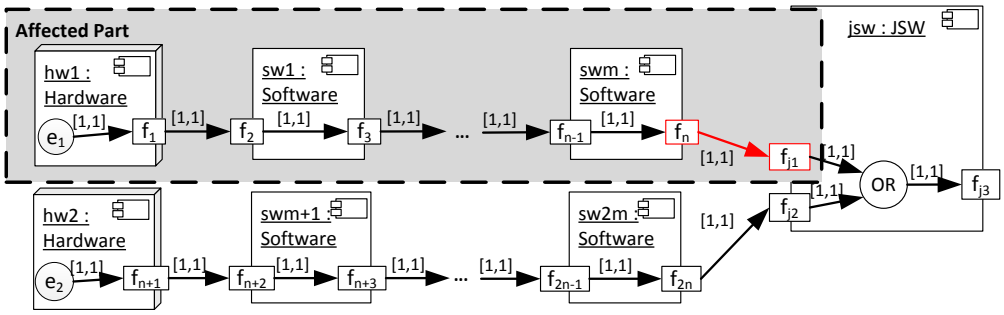


Figure 7.22: TFPG with sequential structure

Figure 7.23 shows a diagram representing the runtimes for configurations with two paths depending on the length of the paths: one affected and one non-affected path. The blue line shows the runtimes of the analysis on the affected subgraph. The red line shows the runtimes for the reachability analysis when applied to the whole system. Table 7.1 shows the numerical results of this experiment. The curve shapes indicate exponential complexity in both cases. However, the blue curve is significantly flatter than the red curve. For example, it took 35.3 hours to analyze a path of length 70 on the full graph and only 2.3 hours using the affected graph. Thus, the computation of the reachability analysis on the affected part increases the feasibility of the approach significantly.

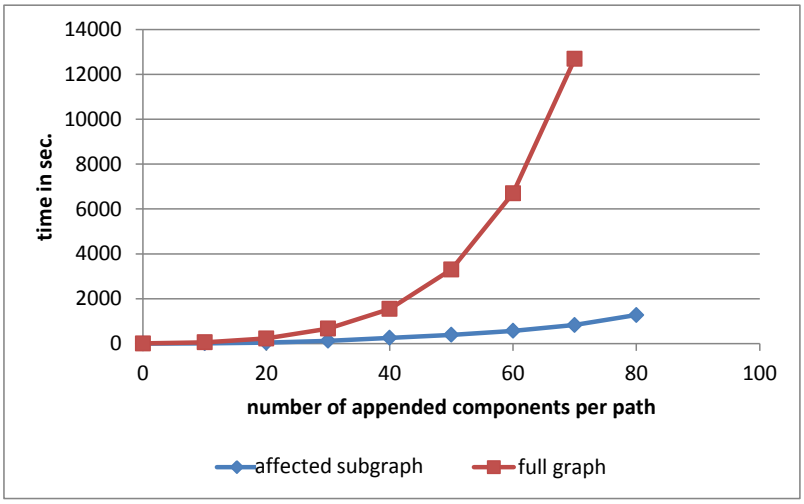


Figure 7.23: Affected vs. full graph

# of TFPG nodes	path length	runtime in sec.	
		affected graph	full graph
6	0	3.249	6.989
46	10	14.226	54.918
86	20	42.785	230.597
126	30	125.344	667.827
166	40	254.585	1543.021
206	50	387.396	3301.724
246	60	566.993	6702.498
286	70	828.591	12687.378
326	80	1269.779	NA

Table 7.1: Evaluation results of the linear structure of Figure 7.22

Tree Structure

In the course of the student's project "SafeBots II", AShOp has been evaluated for TFPGs with binary tree structures as shown in Figure 7.24. All component instances in the configuration have a TFPG with one outgoing failure and two incoming failures, which are connected to the outgoing failure by an operator node. The hardware nodes have one error, which is connected to an outgoing failure. We divide the tree structure into l layers. The first and the $l - 1^{th}$ layer consist of component instances that have TFPGs with an OR operator. During each run of the experiment, we insert a new layer directly after the first layer. We increase the number of component instances and hardware nodes of the layers 3 to l such that the structure of a binary tree is preserved. The new layer consists of component instances that have a TFPG with an AND operator. The experiment starts with the layers 1, $l - 1$, and l.

Table 7.2 shows the results of the experiment. The analysis was able to analyze the tree structure up to a depth of three when analyzing the affected part and a depth of two when analyzing the full graph. For greater tree depths, the memory of the computer was not sufficient.

# of TFPG nodes	# of AND-layers	runtime in sec.	
		affected graph	full graph
20	0	2	4
44	1	4	9,491
76	2	20,386	NA

Table 7.2: Evaluation results for the tree structure of Figure 7.24

Discussion

We compare the results of the latter experiment (Exp. 2) to the results of the experiment using the linear structure of Figure 7.22 (Exp. 1). We do this by

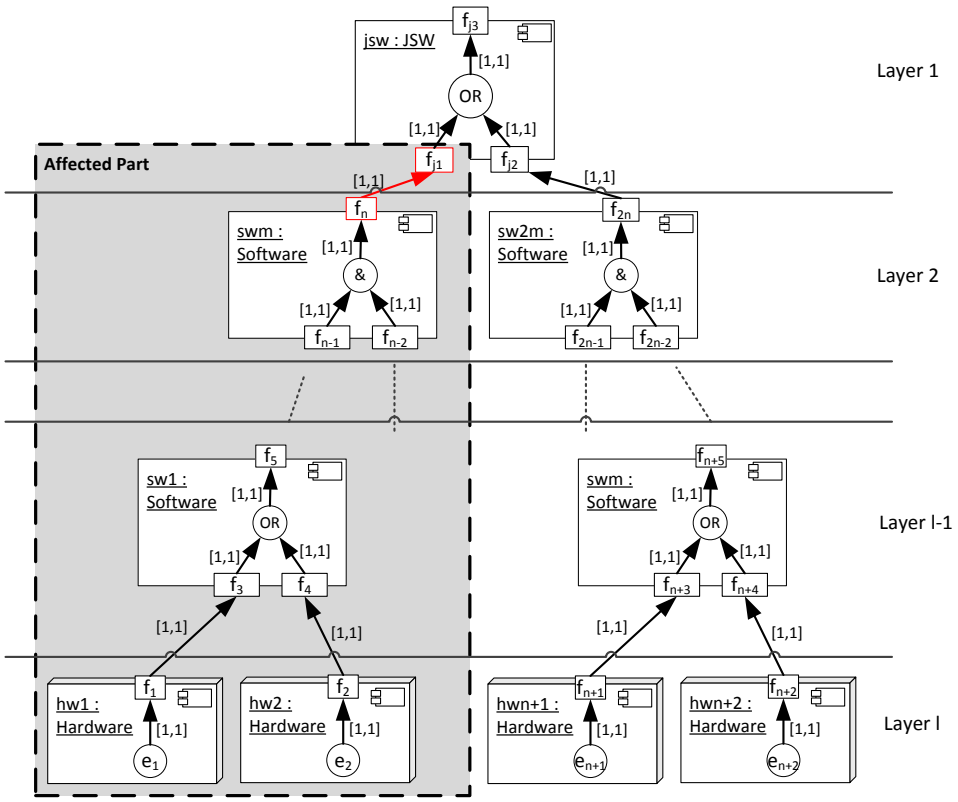


Figure 7.24: TFPG with tree structure

comparing the number of TFPG nodes, which could be analyzed in both experiments. In Exp. 1, we were able to analyze a TFPG with approximately 326 nodes, which corresponds to a path length of 70 (cf. Table 7.1). In Exp. 2, we were able the analyze a TFPG with 76 nodes, which corresponds to two AND-layers (cf. Table 7.2). In Exp. 2, the next step would have been to analyze a TFPG with three AND-layers, which would have contained 140 nodes. In contrast to the linear structure used in Exp. 1, we could not finish the analysis for 140 nodes on the tree structure in Exp. 2 due to insufficient memory. The runtime of the analysis was significantly higher, as well. It took approximately 57 hours to analyze a tree with 76 nodes (depth 3) and only approximately four minutes to analyze a linear structure with 86 nodes (path length 20). Thus, the theoretically exponential algorithm is applicable to TFPGs with approximately 80 nodes and a high degree of branching and to TFPG with approximately 320 nodes and a low degree of branching.

7.4 Summary

In this chapter, we presented the tools, which implement the approaches that we developed in the course of this thesis. We realized the implementation as

a set of plugins for the Fujaba Real-time Tool Suite, which already supported modeling and analyzing software with MechatronicUML.

We first explained the usage of the tools in Section 7.1 as they have been applied to the speed control subsystem of the RailCab. We then gave an overview of their architecture.

Finally, we evaluated our implementation in three case studies. The case studies indicated that the generation of TFPGs and the analysis of self-healing operations has exponential complexity in theory. However, these approaches can still be applied to real systems, because the most complex computations are only applied to relevant parts of the system.

The most complex part in the generation of TFPGs is the computation of the reachable behavior. However, for identifying outgoing timing and service failures, only single paths in the zone graphs of the reachable behavior need to be analyzed (cf. Section 4.2.1). Our evaluation showed that our implementation is able to generate TFPGs from paths with a maximum number of input messages of seven (on the used computer). Thus, all real-time statecharts that have less than seven input messages in each path, may be used to generate TFPGs.

For the evaluation of the analysis of self-healing operations, we focused on the effect of only using the affected part for the reachability analysis. Therefore, we executed the analysis on the complete system and only on the part, which was affected by the self-healing operation. In both cases, the complexity of the analysis theoretically exponential. However, the application of the analysis on the affected part of the system showed a significant improvement compared to the analysis on the whole system. Thus, the reduction of the analyzed model, which only takes the affected part of the system into account, makes the analysis applicable.

8 Related Work

This chapter covers the works of others, which are related to the methods, which have been developed in the course of this thesis. We divide the related work into three categories: First, we review related work for the analysis of self-healing operations in Section 8.1. We focus on approaches that analyze hazard probabilities in reconfigurable systems. In Section 8.2, we consider approaches for the generation of failure propagation models. Since there exists no approach that generates failure propagation models that contain propagation times, we present approaches that generate failure propagation models without propagation times from behavior models. Finally, we describe related work in the field of runtime analysis in Section 8.3. We focus on runtime analyses that pro-actively prevent unsafe system states, like our approach for online analysis.

8.1 AShOp

AShOp is designed for mechatronic systems that react to a detected failure by structural reconfiguration. Therefore, AShOp considers real-time properties and structural reconfiguration. Since there exist no approaches for analyzing hazard occurrence probabilities that consider reconfiguration and propagation times at the same time, we divide the related work into hazard analysis approaches, which consider reconfiguration and hazard analysis approaches which consider reconfiguration and time.

There exists a variety of approaches, which take time into account but no reconfiguration [GCW07, CGW08, GG06, KGF07, KNP11, MS02]. Since we consider the domain of reconfigurable systems, we do not discuss these approaches any further. Instead, we focus on approaches that deal with hazard probabilities in reconfigurable systems. In this field, there exist three approaches: the deductive cause consequence analysis for self-adaptive systems of Güdemann et al. [GOR06], the LARES approach of Walter et al. [WGR+09], and the component-based hazard analysis for reconfigurable systems of Giese et al. [GT06].

8.1.1 Deductive Cause Consequence Analysis for Self-adaptive Systems

Güdemann et al. [GOR06] model the system behavior by a set of automata. Apart from the functional behavior, reconfiguration is modeled by an extra automaton where sets of states are associated to system components. The states specify different behaviors of the system component. The system is reconfigured by switching the states. This means, the behavior of the components is changed. Valid configurations are specified by LTL-formulas. This allows for verifying the specified reconfigurations.

Hazards are specified by predicate formulas. Failures are modeled by automata that define how and when a failure occurs and by predicate formulas modeling the effect of the failure. Minimal cut sets are identified by model checking. Based on these minimal cut sets hazard occurrence probabilities are computed. For reconfigurable systems, the analysis checks whether there always exists a path that leads from a state, in which a minimal cut set is active, back to a safe system state [GOR06].

Self-healing is implemented by the restore invariant approach [NSS+11]. Invariants in the form of predicate formulas specify unwanted system states. Every time an invariant is violated, the system reconfigures autonomously into a state where all invariants are fulfilled. The deductive cause consequence analysis of self-adaptive systems is used to show that the system can always be returned to a safe state.

8.1.2 LARES

The LAnguage for REconfigurable dependable Systems (LARES) of Walter et al. [WGR+09] is a language for modeling fault tolerant systems. It supports modeling repairable systems with hierarchical architectures, common cause errors, and dependabilities between basic events. It allows for computing the probability that the whole system fails even though repair operations are executed. Thereby, the analysis also takes into account at which probability the repair operations may fail.

Architecture is defined text-based as program code. System behavior, including reconfiguration, is state-based. Repair is modeled by transitions between states that represent that a system component is broken and that a system component is working correctly. State transitions are also used for switching system components on and off which, in turn, allows for switching off broken components to repair the system. This also allows for specifying a probability that the repair operation will fail.

For analysis, the LARES models may be transformed into several formal models [RS12], for example stochastic time Petri nets [Zim10] or process algebras [KSW04]. It is thereby possible to compute the probability that the whole system fails and the probability that a repair operation fails.

8.1.3 Component-based Hazard Analysis for Reconfigurable Systems

The component-based hazard analysis for reconfigurable systems of Giese et al. [GT06] computes hazard occurrence probabilities of all architectures of a reconfigurable system. It has been explained in detail in Section 2.4.2.

UML deployment diagrams are extended by hardware ports to specify and analyze the impact of hardware faults on software components. Component types and hardware nodes are enhanced by Boolean formulas that specify the failure propagation inside the component. The failure propagation between hardware nodes and component instances and between component instances is derived automatically from the connectors of the deployment. Hazards are specified by fault trees. The leaves of the fault tree correspond to outgoing failures of the components of the system structure. Qualitative and quantitative fault tree analysis is performed based on Binary Decision Diagrams.

8.1.4 Hybrid Failure Propagation Graphs

The approach of Dubey et al. [DKM11] diagnoses system faults based on alarms raised by the system during runtime. In contrast to our approach, the approach focuses on fault diagnosis. This means, the causing faults are identified.

However, Dubey et al. [DKM11] use a model called hybrid failure propagation graphs. As in our TFPGs (cf. Section 3.4), the edges in hybrid failure propagation graphs have time intervals that specify the propagation times of failures. Hybrid failure propagation graphs differ from our TFPGs in some details: failures, which are called discrepancies, and operators are modeled in one node. Edges can be activated and deactivated to model reconfigurable systems.

We use the syntax of fault trees [VGRH81] and enhance it by the propagation time intervals, because fault trees are commonly used in literature.

8.1.5 Discussion

The approaches of Güdemann et al. [GOR06], Walter et al. [WGR+09], and Giese et al. [GT06] presented above do not consider time. The restore invariant approach [NSS+11], which analyzes self-healing systems does not consider time, as well. It can therefore only analyze whether there is a path in the system behavior that leads to a healthy system state in case of a hazard. However, the analysis cannot judge whether the self-healing is executed fast enough.

The component-based hazard analysis for reconfigurable systems [GT06] computes hazard occurrence probabilities for single architectures. Since it does not consider propagation times of failures, it cannot compute the locations of failures in the system at the point of time when the reconfiguration is applied.

Consequently, the approach cannot take into account how the structural reconfiguration affects the propagation of failures, which are present in the system. It is thus not able to assess the success of self-healing operations.

All approaches support the computation of hazard probabilities in reconfigurable systems. While Güdemann et al. [GOR06] take the whole system behavior into account, Walter et al. [WGR+09] use an abstracted behavior model, and Giese et al. [GT06] work on failure propagation models. Consequently, the analyses of Walter et al. [WGR+09] and Giese et al. [GT06] do not need to explore the whole system behavior. They only take the behavior into account, which is relevant for failure propagation. On the other hand, the deductive cause consequence analysis for self-adaptive systems [GT06] is more precise.

We chose to also use failure propagation models for AShOp and extend them by time intervals as Dubey et al. [DKM11]. But instead of constructing them manually as in [GT06, DKM11], we generate them from real-time behavior models (cf. Section 4). This means, we also take the complete system behavior into account. However, for AShOp we abstract from information, which is not needed for hazard analysis. We only consider the failure propagation over time and do not need to evaluate behavior like variable assignments or silent transitions.

All approaches differ in the specification of reconfiguration, as well. Güdemann et al. [GOR06] and Walter et al. [WGR+09] model reconfiguration by state switches and predicate formulas while Giese et al. [GT06] use structural reconfiguration. In contrast to simple state switches, structural reconfiguration preserves all advantages of component based modeling like encapsulation, decomposition and hierarchical architecture (cf. Section 2.2.3). Our analysis therefore also operates on structural reconfiguration.

8.2 Automatic Generation of Failure Propagation Models

There exists a wide range of approaches, which are able to generate fault trees that do not contain timing information. The source models for the fault trees are for example behavior models [BV07, KLFL11, LR98, MPW10, Rau02], AADL[1] models [DD08, JBV07, LZMX11], or decision tables [HA97].

The approaches that generate fault trees from behavior models provide the foundations for our generation of TFPGs from real-time statecharts (cf. Chapter 4). These approaches use different behavior models as source models, for example state machines [LR98, MPW10], continuous time Markov chains [KLFL11], NuSMV models [BV07], or mode automata [Rau02]. All of these approaches use reachability analysis, mostly model checking, and follow a similar process: The failures are modeled as part of the behavior model. This is called “fault

[1] Architecture Analysis & Design Language [FG12]

injection". The model checker is used to construct the reachable behavior of the system with injected faults. Failures are identified in this reachable behavior. The fault tree is constructed from the identified and injected failures.

8.2.1 Continuous Time Markov Chains

Most of the approaches, which use model checking, do not take timing information into account. The only approach which considers time is the approach of Kuntz et al. [KLFL11]. However, this approach does not compute the minimum and maximum runtime of paths, which are needed for AShOp.

Kuntz et al. [KLFL11] generate fault trees from continuous time Markov chains (CTMC) using the probabilistic model checker PRISM [KNP11]. A Markov chain is a stochastic process with a discrete state space [CL06]. Transitions fire with a specified probability. In CTMCs, the probabilities are distributed exponentially over a continuous time interval. The goal of probabilistic model checking is to check the probability of being in any system state at a certain point in time [CL06].

Kuntz et al. [KLFL11] use the counterexamples of the probabilistic model checker PRISM [KNP11] to construct a fault tree. However, it is not possible to compute propagation times from these counterexamples, because the timing information in CTMC does not define firing times of transitions. Instead, transitions are labeled with firing rates, which specify the probability of a state change within a specified time interval. Thus, the generated fault trees do not contain propagation times.

8.2.2 State Machines

Liggesmeyer et al. [LR98] provide an approach for the automatic generation of fault trees from finite state machines. Failures are injected into state machines first by adding transitions that model failures. Then, the reachable behavior of the original state machine and the state machine with failures is compared. A failure model specifies which failures should be considered.

Failures are all values that are specified for a device. Therefore, the set of failures can be generated automatically from a component specification.

First, the set of all failures that may lead to an undesired event is generated. For this, the reachability analysis of a model checker is applied. Fault trees are constructed according to Vesely et al. [VGRH81] from the reachable behavior of the failure-free system and the behavior of the system with failures.

Mahmud et al. [MPW10] present an approach for the generation of temporal fault trees from state machines. The generation is performed by a backward search from the final states of the state machine back to the initial state. The final states are the top events of the temporal fault tree. Each path from a final state to the initial state is a branch in the fault tree.

8.2.3 Mode Automata

Rauzy [Rau02] presents an approach to generate fault trees from mode automata. In mode automata, states are modes. A mode specifies an active transfer function that determines how to process system inputs. Events trigger transitions and are used to model failures.

For the generation of fault trees, failures are represented by events of transitions. Some modes of states represent failure modes. All paths from the initial state to a failure mode are failure scenarios. To get these, the reachable behavior of the mode automaton is constructed and the failure scenarios are mapped to Boolean formulas. These, in turn, are represented as fault trees.

8.2.4 FSAP/NuSMV-SA

The model checker NuSMV is used to generate fault trees from a model given in the NuSMV input language [BV07] and a top-level event. The goal of the analysis is to extract minimal cut sets, which are then represented as fault trees. For this, faults are injected into the NuSMV input model. On this model, a forward reachability is performed which yields all reachable states in which the top-level event occurs. Intermediate events are removed by exist quantification [Rau03]. The minimal cut sets are computed by evaluating a binary decision diagram as described by Giese et al. [GTS04].

8.2.5 Discussion

None of the presented approaches computes propagation times of failures. Thus, the generated failure propagation models cannot be used by AShOp. Instead, we adapt the approach of Liggesmeyer et al. [LR98] to generate TFPGs, because the input models are most similar to our real-time statecharts.

8.3 Runtime Analysis

In Chapter 6, we presented an approach to analyze self-healing operations at runtime. There exists a wide range of approaches for runtime analysis [DDK+07, FGT11, GMS12, KMM07, Rus08, SBT11, SRA04]. The only approach which is closely related to our runtime analysis is the runtime certification of Schneider et al. [SBT11].

Another field in runtime analysis is models at runtime [GLB+12, KCF12, MLK12, NFPB12, VG10]. Models, which are used at design time, are also used at runtime. However, at runtime, models are required, which provide views related to the problem spaces, i.e., they only contain information, which is necessary for the specific analysis. We abstract from information, which is unnecessary for AShOp, as well, by generating TFPGs from real-time statecharts.

8.3.1 Runtime Certification

Schneider et al. [SBT11] propose an approach for runtime certification. They consider systems of systems where different systems are combined at runtime to provide higher-level functionalities. These combinations have to be certified at runtime and before the systems are combined, because they cannot be foreseen at design time.

Runtime certification is based on conditional certificates that are created at design time and specify which properties a system demands and guarantees. The idea is to perform the actual certification at runtime but to conduct complex interpretation steps at design time. Systems are only combined at runtime if the combination can be certified.

Runtime certification is the only approach that analyzes configurations at runtime before they are constructed. However, Schneider et al. [SBT11] do not focus on executing any particular analysis at runtime as the certificates can be used by any safety analysis. As a consequence, Schneider et al. [SBT11] do not address the analysis of self-healing operations at runtime. However, our analysis may be integrated into runtime certification.

8.3.2 Other Approaches for Runtime Analysis

Apart from runtime certification, there exist approaches for runtime verification and quality service management in service-based systems.

Approaches for runtime verification [DDK+07, FGT11, GMS12, KMM07, Rus08, SRA04] focus on the detection of anomalies in the executed behavior of the system and try to lead the system back to its intended behavior. Anomalies are detected by monitors, which check the executed program continuously against a specification. In contrast to runtime verification, our analysis aims at preventing unsafe system states.

A wide range of approaches for quality management in service-based systems is summarized in the survey of Calinescu et al. [CGK+11]. A quality management identifies and enforces optimal system configuration to ensure quality of service requirements by adapting to changes in the system. Again, quality management is a reaction to anomalous system behavior, while our analysis aims at avoiding such behavior.

9 Conclusion

9.1 Summary

We find a growing number of technical systems in our world today. They facilitate our lives by performing tedious and exhausting tasks like housework. Technical systems like air planes or high-tech trains make the world seem smaller by letting us move longer distances in shorter times. However, these benefits involve negatives aspects, as well. Whenever a technical device is operating, random faults may occur and cause hazards, which may lead to accidents like electric shocks or crashes.

Developers identify design faults by applying a systematic and structured development process, for example a model-based development. However, random faults that occur due to hardware flaws cannot be avoided completely. To still construct a safe system, the developer must guarantee that random faults and resulting hazards only occur with an acceptable probability.

One way to achieve acceptable hazard occurrence probabilities is self-healing. Self-healing systems react autonomously to observed failures by removing these failures or cutting of failure propagation paths that lead to hazards. In this work, self-healing operations are implemented by structural reconfigurations, which create or remove components from the system or modify connections between components at runtime.

Of course, the developer must guarantee that the application of a self-healing operation actually reduces the occurrence probability of a hazard to an acceptable value. In the course of this thesis, we developed an approach to analyze the effect of self-healing operations in mechatronic systems (AShOp) which has been introduced in Chapter 5. AShOp takes in particular the propagation times of failures into account. This allows for analyzing how far failures propagate through the system within a specific time interval. This, in turn, allows for computing the locations of errors and failures in the system at the point in time when the self-healing operation is executed which then enables to analyze which errors and failures are removed from the system or stopped from propagating.

Such analyses are usually carried out at design time. However, in reconfigurable systems, configurations may occur at runtime that have been unknown at design time. Still, the developer must guarantee acceptable hazard occurrence probabilities. Therefore, we presented a framework that allows for executing AShOp at runtime in Chapter 6. Future configurations, which would exceed acceptable

hazard occurrence probabilities even with the application of self-healing operations, are locked. Consequently, the system will only create configurations with acceptable hazard occurrence probabilities.

To enable this analysis, we introduced timed failure propagation graphs (TFPGs) in Chapter 3. TFPGs are failure propagation models, which are extended by propagation times of failures. We defined formal semantics for TFPGs to allow for the computation of the propagation of failures over time. TFPGs are generated automatically from real-time statecharts (cf. Chapter 4) that specify the behavior of the component types of the system.

The methods and models which have been developed in the course of this thesis have been implemented as plugins for the Fujaba Real-time Tool Suite as described in Chapter 7. We applied the implementation to analyze the RailCab and conducted experiments to analyze the scalability of the generation of TFPGs and AShOp.

9.2 Future Work

The propagation time intervals of TFPGs allow for computing failure propagation times as presented in Chapter 3. However, the usage of intervals also adds uncertainty to the propagation times: The longer the path in the TFPG, which is analyzed, the wider the computed propagation time interval of the path will become. This uncertainty may be relaxed by adding probability distributions to the time intervals. This would allow for analyzing for example the most probable or average failure propagation times of a TFPG path. This can be achieved, for example, by adding probability distributions to the propagation time intervals of TFPGs and using deterministic stochastic time Petri nets (DSPN) as a formal semantics. DSPNs allow for analyzing stochastic delays and probabilities of decisions [Zim08]. Another option is to use stochastic timed automata [CL06]. However, if systems consist of many communicating stochastic timed automata, the analysis will suffer from state space explosion.

Our analysis of self-healing operations may be extended by taking the durations of error occurrences into account, because hazard occurrence probabilities also depend on how long an error is present [Lev95].

AShOp takes a configuration and a self-healing operation as input and analyzes the success of the self-healing operation. The self-healing operation is constructed manually. AShOp could also be used reversely to automatically generate self-healing operations from a given configuration including the earliest and latest time of application of the self-healing operation. The generated self-healing operation would already guarantee to successfully reduce the occurrence probability of a hazard to an acceptable value.

AShOp analyzes the real-time statecharts that model the system behavior to compute the delay between the failure detection and the execution of the self-healing operation. This analysis requires the computation of the reachable

state space of the complete system, which may suffer from state space explosion. Future research should work on decomposing the system to address this problem by exploiting existing compositional verification approaches.

Our TFPG generation as presented in Chapter 4 computes relations between incoming and outgoing timing and service failures and relations between incoming and outgoing value failures. Relations between value failures and service failures or between value failures and timing failures cannot be identified, because we use different methods to compute this relation. Of course, these relations exist and our methods for the identification of value failures and of service and timing failures should be combined such that the missing relation can be identified.

Further, our TFPG generation is currently not able to evaluate side effects of transitions. Instead, we assume that all variables that are used as input parameters of side effects have an effect on the output of the side effect. This is an over approximation which may lead to the computation of too high hazard occurrence probabilities. Therefore, future works should consider the evaluation of side effects.

Future research should also investigate how components can be divided into subcomponents such that the timed automaton of each subcomponent only has a limited amount of transitions with input messages. In this way, the complexity of the TFPG generation can be reduced such that TFPGs can also be generated for timed automata with paths that have many inputs.

Our approach for the runtime analysis uses an analyzer component that conducts the analysis. However, this introduces a single point of failure. If the analyzer fails, the analysis might output wrong results and the system might reconfigure into an unsafe configuration. To avoid a single point of failure, the analyzer may be executed in several subsystems and the results could by compared.

Another field is a concept for a compositional analysis of self-healing operations. Each autonomous component computes its hazard probabilities by itself and the autonomous components only exchange hazard probabilities instead of behavioral models. Thus, the computing load is distributed among the whole system. We already developed a decomposition of the hazard analysis of Tichy et al. [GT06] in the master's thesis of Anis [Ani12].

The failure propagation models of Tichy et al. [GT06] support the reconfiguration of communication links. However, the failure propagation inside a component is static. Consequently, the failure propagation of each component instance needs to be specified explicitly. This causes a great amount of models to be constructed manually. In the case of multi-ports, this is even not applicable, because multi-ports have a varying number of subports. We suggest using dynamic failure propagation models for online hazard analysis. First ideas have been worked out by Braun [Bra12]. Dynamic failure propagation models are specified for component types and adapt according to the component instance. For multi-ports, a minimum number of incoming failures of each type is identified such that a failure is propagated.

For mechatronic systems, the union of four disciplines into one system requires the development and analysis of the system as a whole. The key difference to pure software architecture is that (hardware) connectors that are connected to hardware components do not only transport information but also physical items, i.e. material and energy. The active structure is a model that specifies the architecture of the entire mechatronic system including hardware and software parts. Hardware connectors are only represented as simple connections, even though they represent additional system components. In [PSTH11] we presented a component-based hazard analysis that considers the entire mechatronic system including hardware connectors and introduces reusable patterns for the failure behavior of hardware connectors, which can be generated automatically. In this way, the component-based hazard analysis of [GT06] can be applied to the entire mechatronic system.

List of Abbreviations

AShOp analysis of self-healing operations 5

CRC 614 Collaborative Research Center 614 “Self-optimizing concepts and structures in mechanical engineering” 16

CTMC continuous time Markov chain 155

DSPN deterministic stochastic Petri net 160

ECU electronic control unit 24

EFSM extended finite state machine 81

MCS minimal cut set 35

MTTF mean time to failure 35

NTA network of timed automata 50

NTAC network of timed automata that specifies the behavior of a component type 51

OCM Operator-Controller-Module 18

SIL safety integrity level 40

TCSD timed component story diagram 28

TCSP timed component story pattern 28

TFPG timed failure propagation graph 43

TPN time Petri net 58

WCET worst case execution time 120

Own Publications

[BBD+12] Steffen Becker, Christian Brenner, Stefan Dziwok, Thomas Gewering, Christian Heinzemann, Uwe Pohlmann, Claudia Priesterjahn, Wilhelm Schäfer, Julian Suck, Oliver Sudmann, and Matthias Tichy. The MechatronicUML method – process, syntax, and semantics. Technical Report tr-ri-12-318, Software Engineering Group, University of Paderborn, feb 2012.

[BDG+11] S. Becker, S. Dziwok, T. Gewering, C. Heinzemann, U. Pohlmann, C. Priesterjahn, W. Schäfer, O. Sudmann, and M. Tichy. Mechatronicuml - syntax and semantics. Technical report, Software Engineering Group, Heinz Nixdorf Institute, 2011.

[EHH+13] Tobias Eckardt, Christian Heinzemann, Stefan Henkler, Martin Hirsch, Claudia Priesterjahn, and Wilhelm Schäfer. Modeling and verifying dynamic communication structures based on graph transformations. *Comput. Sci.*, 28(1):3–22, February 2013.

[HPB12] C. Heinzemann, C. Priesterjahn, and S. Becker. Towards modeling reconfiguration in hierarchical component architectures. In *15th ACM SigSoft International Symposium on Component-Based Software Engineering (CBSE 2012)*, 2012.

[PHS13] Claudia Priesterjahn, Christian Heinzemann, and Wilhelm Schäfer. From timed automata to timed failure propagation graphs. In *Proceedings of the Fourth IEEE Workshop on Self-Organizing Real-time Systems*, 2013. accepted.

[PHST12] C. Priesterjahn, C. Heinzemann, W. Schäfer, and M. Tichy. Runtime safety analysis for safe reconfiguration. In *Proceedings of the 3. Workshop „Self-X and Autonomous Control in Engineering Applications", 10. IEEE International Conference on Industrial Informatics, 25. – 27. Juli 2012, Beijing, China*, 2012.

[PST11] Claudia Priesterjahn, Dominik Steenken, and Matthias Tichy. Component-based timed hazard analysis of self-healing systems. In *Proceedings of the 8th workshop on Assurances for self-adaptive systems*, ASAS '11, pages 34–43, New York, NY, USA, 2011. ACM.

[PST13] Claudia Priesterjahn, Dominik Steenken, and Matthias Tichy. Timed hazard analysis of self-healing systems. In Javier Camara, Rogerio de Lemos, Carlo Ghezzi, and Antonia Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *LNCS*, pages 112–151. Springer Berlin Heidelberg, 2013.

[PSTH11] Claudia Priesterjahn, Christoph Sondermann-Wölke, Matthias Tichy, and Christian Hölscher. Component-based hazard analysis for mechatronic systems. *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops , IEEE International Symposium on*, pages 80–87, 2011.

[PT09] C. Priesterjahn and M. Tichy. Modeling safe reconfiguration with the fujaba real-time tool suite. In *Proceedings of the 7th International Fujaba Days*, 2009.

[PTH+10] C. Priesterjahn, M. Tichy, S. Henkler, M. Hirsch, and W. Schäfer. Fujaba4eclipse real-time tool suite. In *Model-Based Engineering of Embedded Real-Time Systems (MBEERTS)*. Springer, 2010.

References

[AAB+11] Amir Shayan Ahmadian, Caner Aydogan, Denis Braun, Luis G. Bustamante, Christopher Gerking, Süleyman Issiz, Lukas Kopecki, and Paul Prescher. Developer documentation of the project group safebots i. Project group, University of Paderborn, Department of Computer Science, Paderborn, Germany, September 2011.

[ACH+12] Kelly Androutsopoulos, David Clark, Mark Harman, Robert M. Hierons, Zheng Li, and Laurence Tratt. Amorphous slicing of extended finite state machines. *IEEE Transactions on Software Engineering*, 99(PrePrints):1, 2012.

[ADG+09] Philipp Adelt, Jörg Donoth, Jens Geisler, Stefan Henkler, Sascha Kahl, Benjamin Klöpper, Eckehard Münch, Simon Oberthür, Carlos Paiz, Herbert Podlogar, Mario Porrmann, Rafael Radkowski, Christoph Romaus, Alexander Schmidt, Bernd Schulz, Henner Voecking, Ulf Witkowski, and Katrin Witting. *Selbstoptimierende Systeme des Maschinenbaus – Definitionen, Anwendungen, Konzepte*. Number 234, in HNI Verlagsschriftenreihe. Heinz Nixdorf Institute, University of Paderborn, 2009.

[AGL+12] Anas Anis, Sebastian Goschin, Sebastian Lehrig, Christian Stritzke, and Thomas Zolynski. Developer documentation of the project group safebots ii. Project group, University of Paderborn, Department of Computer Science, Paderborn, Germany, March 2012.

[AH99] Karl J. Astrtöm and Tore Hägglund. Pid control. In William S. Levine, editor, *The Control Handbook*, chapter 10.5. Jaico Publishing House, Mumbai, India, 1999.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[Alu99] Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron A. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99), July 6-10, 1999, Trento, Italy*, volume 1633 of *Lecture Notes in Computer Science (LNCS)*, pages 8–22. Springer Verlag, 1999.

[Ani12] Anas Anis. Component-based decomposition of hazard analysis. Master's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, 2012.

[AO02] B. Arslan and A. Orailoglu. Fault dictionary size reduction through test response superposition. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 480 – 485, 2002.

[BBD+07] S. Bentler, S. Brügger, R. Dorociak, T. Hoffmann, J. Holtmann, W. Janzen, M. Knoop, A. Oberhoff, S. Polat, R. Reinert, M. von Detten, and C. Werner. *Abschlussarbeit der Projektgruppe ASE : Automotives Software Engineering.* Project group, Fakultät für Elektrotechnik, Informatik und Mathematik, Universität Paderborn, 2007.

[BBI+11] Danny Bickson, Dror Baron, Alexander T. Ihler, Harel Avissar, and Danny Dolev. Fault identification via nonparametric belief propagation. *IEEE Transactions on Signal Processing*, 59(6):2602–2613, 2011.

[BD09] V.S. Bagad and I.A. Dhotre. *Data Communication & Networking.* Technical Publications, 2009.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, sep 2004.

[BDL+06] Gerd Behrmann, Alexandre David, Kim G. Larsen, Paul Pettersson, Wang Yi, and Martijn Hendriks. Uppaal 4.0. In *In Quantitative Evaluation of Systems - (QEST'06*, pages 125–126. IEEE Computer Society, 2006.

[BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental design and formal verification with uml/rt in the fujaba real-time tool suite. In *Proc. of the Intern. Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004*, pages 1–20, Oct. 2004.

[BGK+96] Johan Bengtsson, David W. O. Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an audio protocol with bus collision using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *cav96*, number 1102 in LNCS, pages 244–256. Springer–Verlag, July 1996.

[BHF96] Vamsi Boppana, Ismed Hartanto, and W. Kent Fuchs. Full fault dictionary storage based on labeled tree encoding. In *in 14th IEEE VLSI Test Symposium (VTS'96), April 28 - May 1*, pages 174–179, 1996.

[BO10] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

[Bra12] Denis Braun. Erweiterung eines gefahrenanalyse-ansatzes für rekonfigurierbare systeme. Master's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, 2012.

[BSMH84] Robert K. Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[BV07] Marco Bozzano and Adolfo Villafiorita. The fsap/nusmv-sa safety analysis platform. *Int. J. Softw. Tools Technol. Transf.*, 9(1):5–24, February 2007.

[BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[CGK+11] R. Calinescu, Lars Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *Software Engineering, IEEE Transactions on*, 37(3):387–409, 2011.

[CGKM12] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, and Raffaela Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, September 2012.

[CGP08] Antonio Carzaniga, Alessandra Gorla, and Mauro Pezzè. Self-healing by means of automatic workarounds. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, pages 17–24, New York, NY, USA, 2008. ACM.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[CGW08] Robert Colvin, Lars Grunske, and Kirsten Winter. Timed behavior trees for failure mode and effects analysis of time-critical systems. *J. Syst. Softw.*, 81:2163–2182, December 2008.

[CK10] Radu Calinescu and Marta Z. Kwiatkowska. Software engineering techniques for the development of systems of systems. In *Foundations of Computer Software. Future Trends and Techniques for Development, 15th Monterey Workshop 2008, Budapest, Hungary, September 24-26, 2008, Revised Selected Papers*, volume 6028 of *Lecture Notes in Computer Science*, pages 59–82. Springer, 2010.

[CKKK06] Sunghoon Chun, Sangwook Kim, Hong-Sik Kim, and Sungho Kang. An efficient dictionary organization for maximum diagnosis. *J. Electron. Test.*, 22(1):37–48, February 2006.

[CL06] Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 2006.

[Cla95] D. W. Clarke. Sensor, actuator, and loop validations. *IEE Control Systems*, 15:39–45, 8 1995.

[Com98] International Electrotechnical Commission. International Standard IEC 61508. functional safety of electrical/electronic/programmable electronic safety-related systems, 1998.

[CR05] Franck Cassez and Olivier-H. Roux. Structural translation from time petri nets to timed automata. *Electron. Notes Theor. Comput. Sci.*, 128:145–160, May 2005.

[DD08] Josh Dehlinger and Joanne Bechta Dugan. Analyzing dynamic fault trees derived from model-based system architectures. *Nuclear Engineering and Technology*, 40(5), 8 2008.

[DDK+07] Christoph Danne, Viktor Dück, Benjamin Klöpper, Jürgen Brinkmann, and Matthias Tichy. Considering runtime restrictions in self-healing distributed systems. In *Proceedings of the IEEE 21st International Conference on Advanced Information Networking and Applications (AINA-07), Niagara Falls, Canada*. IEEE Computer Society Press, May 2007.

[DKM11] A. Dubey, G. Karsai, and N. Mahadevan. Model-based software health management for real-time systems. In *Aerospace Conference, 2011 IEEE*, pages 1 –18, march 2011.

[dKMR92] Johan de Kler, Alan K. Mackworth, and Raymond Reiter. Characterizing dianosis and systems. *Artifical Intelligence*, 56, 1992.

[dKW87] J de Kleer and B C Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, April 1987.

[dLGB+10] Juan de Lara, Esther Guerra, Artur Boronat, Reiko Heckel, and Paolo Torrini. Graph transformation for domain-specific discrete event time simulation. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Graph Transformations*, volume 6372 of *Lecture Notes in Computer Science*, pages 266–281. Springer, 2010.

[Dun02] William R. Dunn. *Practical Design of Safety-Critical Computer Systems*. Reliability Press, 2002.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006.

[EW94] Uffe H. Engberg and Glynn Winskel. Linear logic on petri nets. Technical report, Department of Computer Science, University of Aarhus, 1994.

[EWM90] Uffe Engberg, Glynn Winskel, and Ny Munkegade. Petri nets as models of linear logic. In *Proceedings of Colloquium on Trees in Algebra and Programming*, pages 147–161. Springer-Verlag LNCS, 1990.

[FG12] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 1st edition, 2012.

[FGT11] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 341–350, New York, NY, USA, 2011. ACM.

[FMNP94] P. Fenelon, J. A. McDermid, M. Nicolson, and D. J. Pumfrey. Towards integrated safety analysis and design. *ACM SIGAPP Applied Computing Review*, 2(1):21–32, 1994.

[GB03] Holger Giese and Sven Burmester. Real-Time Statechart Semantics. Technical Report tr-ri-03-239, Lehrstuhl für Softwaretechnik, Universität Paderborn, Paderborn, Germany, June 2003.

[GCW07] Lars Grunske, Robert Colvin, and Kirsten Winter. Probabilistic model-checking support for fmea. *Quantitative Evaluation of Systems, International Conference on*, pages 119–128, 2007.

[GFDK09] Jürgen Gausemeier, U. Frank, J. Donoth, and S. Kahl. Specification technique for the description of self-optimizing mechatronic systems. *Research in Engineering Design*, 20:201–223, 2009.

[GG06] Grzegorz Golaszewski and Janusz Gorski. Hazard prevention by forced time constraints. In *Proceedings of the International Conference on Dependability of Computer Systems*, DEPCOS-RELCOMEX '06, pages 84–91, Washington, DC, USA, 2006. IEEE Computer Society.

[GGS+07] Jürgen Gausemeier, Holger Giese, Wilhelm Schäfer, Björn Axenath, Ursula Frank, Stefan Henkler, Sebastian Pook, and Matthias Tichy. Towards the design of self-optimizing mechatronic systems: Consistency between domain-spanning and domain-specific models. In *Proc. of the 16th International Conference on Engineering Design (ICED)*, Paris, France, 8 2007.

[GLB+12] Holger Giese, Leen Lambers, Basil Becker, Stephan Hildebrandt, Stefan Neumann, Thomas Vogel, and Sebastian Wätzoldt. Graph transformations for mde, adaptation, and models at runtime. In

Alfonso Pierantonio Marco Bernardo, Vittorio Cortellessa, editor, *Formal Methods for Model-Driven Engineering*, volume 7320 of *Lecture Notes in Computer Science (LNCS)*, pages 137–191. Springer, 6 2012.

[gmf12] Graphical modeling project, 2012.

[GMP+10] Alessandra Gorla, Leonardo Mariani, Fabrizio Pastore, Mauro Pezzè, and Jochen Wuttke. Achieving cost-effective software reliability through self-healing. *Computing and Informatics*, 29(1):93–115, 2010.

[GMS12] Carlo Ghezzi, Andrea Mocci, and Mario Sangiorgio. Runtime monitoring of component changes with spy@runtime. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1403–1406, Piscataway, NJ, USA, 2012. IEEE Press.

[GOR06] Matthias Güdemann, Frank Ortmeier, and Wolfgang Reif. Safety and dependability analysis of self-adaptive systems. In *Proccedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006)*, nov 2006.

[Gro09] Object Management Group. Uml 2.2 superstructure specification, 2009.

[GSG+09] Jürgen Gausemeier, Wilhelm Schäfer, Joel Greenyer, Sascha Kahl, Sebastian Pook, and Jan Rieke. Management of cross-domain model consistency during the development of advanced mechatronic systems. In Margareta Norell Bergendahl, Martin Grimheden, and Larry Leifer, editors, *Proc. of the 17th International Conference on Engineering Design (ICED'09)*, volume 6 of *Design Society*, pages 1–12, August 2009.

[GSRU07] Debanjan Ghosh, Raj Sharman, Rao H. Raghav, and Shambhu Upadhyaya. Self-healing systems - survey and synthesis. *Decis. Support Syst.*, 42(4):2164–2185, January 2007.

[GT06] Holger Giese and Matthias Tichy. Component-based hazard analysis: Optimal designs, product lines, and online-reconfiguration. In *Proc. of the 25th International Conference on Computer Safety, Security and Reliability (SAFECOMP), Gdansk, Poland*, Lecture Notes in Computer Science (LNCS), pages 156–169. Springer Verlag, September 2006.

[GTB+03] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time uml designs. In *Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11)*, 2003.

[GTS04] Holger Giese, Matthias Tichy, and Daniela Schilling. Compositional Hazard Analysis of UML Components and Deployment Models. In *Proc. of the 23rd International Conference on Computer Safety, Reliability and Security, Potsdam, Germany*, volume 3219 of *LNCS*. Springer Verlag, September 2004.

[HA97] J.J. Henry and J.D. Andrews. Computerized fault tree construction for a train braking system. *Quality and Reliability Engineering International*, pages 293–298, 1997.

[HC93] M. P. Henry and D. W. Clarke. The self-validating sensor: rationale, definitions, and examples. *Control Engineering Practice*, 1(2):585–610, 1993.

[HSE10] C. Heinzemann, J. Suck, and T. Eckardt. Reachability analysis on timed graph transformation systems. In *Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)*, 2010.

[HSST13] Christian Heinzemann, Oliver Sudmann, Wilhelm Schäfer, and Matthias Tichy. A discipline-spanning development process for self-adaptive mechatronic systems. In *Proceedings of the 2013 International Conference on Software and System Process (ICSSP)*, 2013. accepted.

[Ise07] Rolf Isermann. Fehlertolerante mechatronische systeme, teil 1 (fault-tolerant mechatronic systems, part 1). *Automatisierungstechnik*, 55(4):170–179, 2007.

[JBV07] Anjali Joshi, Pam Binns, and Steve Vestal. Automatic generation of fault trees from aadl models. In *ICSE Workshop on Aerospace Software Engineering, Minneapolis*, 2007.

[Joh03] A. Johnson. Efficient fault analysis in linear analog circuits. *Circuits and Systems, IEEE Transactions on*, 26(7):475–484, January 2003.

[KCF12] Filip Křikava, Philippe Collet, and Robert France. Actor-based Runtime Model of Adaptable Feedback Control Loops. In *International Workshop on models@run.time 2012(MRT)*, Innsbruck, 2012.

[KGF07] B. Kaiser, C. Gramlich, and M. Förster. State/event fault trees–A safety analysis model for software-controlled systems. *Reliability Engineering & System Safety*, 92(11):1521–1537, 2007.

[KLFL11] Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. From probabilistic counterexamples via causality to fault trees. In *30th International Conference on Computer Safety, Reliability and Security (SAFECOMP 2011)*, 2011.

[KMM07] Ingolf Krüger, Michael Meisinger, and Massimiliano Menarini. Runtime verification of interactions: From mscs to aspects. In Oleg Sokolsky and Serdar Tasiran, editors, *Runtime Verification*, volume 4839 of *Lecture Notes in Computer Science*, pages 63–74. Springer Berlin / Heidelberg, 2007.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 1997.

[KSW04] Matthias Kuntz, Markus Siegle, and Edith Werner. Symbolic performance and dependability evaluation with the tool caspa. In Manuel Núñez, Zakaria Maamar, FernandoL. Pelayo, Key Pousttchi, and Fernando Rubio, editors, *Applying Formal Methods: Testing, Performance, and M/E-Commerce*, volume 3236 of *Lecture Notes in Computer Science*, pages 293–307. Springer Berlin Heidelberg, 2004.

[Lev95] Nancy G. Leveson. *Safeware: System Safety and Computers*. ACM, 1995.

[LKN+11] Patrick E. Lanigan, Soila Kavulya, Priya Narasimhan, Thomas E. Fuhrman, and Mutasim A. Salman. Diagnosis in automotive systems: A survey. Technical Report CMU-PDL-11-110, Carnegie Mellon University Parallel Data Lab, June 2011.

[LLC08] Seagate Technology LLC. Data sheet baracuda 7200.11, 2008.

[LR98] P. Liggesmeyer and M. Rothfelder. Improving system reliability with automatic fault tree generation. In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, Washington, DC, USA, 1998. IEEE Computer Society.

[LRST09] Didier Lime, OlivierH. Roux, Charlotte Seidner, and Louis-Marie Traonouez. Romeo: A parametric model-checker for petri nets with stopwatches. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *Lecture Notes in Computer Science*, pages 54–57. Springer Berlin Heidelberg, 2009.

[LZMX11] Yue Li, Yi-an Zhu, Chun-yan Ma, and Meng Xu. A method for constructing fault trees from aadl models. In *Proceedings of the 8th international conference on Autonomic and trusted computing*, ATC'11, pages 243–258, Berlin, Heidelberg, 2011. Springer-Verlag.

[McC56] Edward J. McCluskey. Minimization of Boolean Functions. *Bell System Technical Journal*, 35, 1956.

[MDDW05] M.L. McIntyre, W.E. Dixon, D.M. Dawson, and I.D. Walker. Fault identification for robot manipulators. *Robotics, IEEE Transactions on*, 21(5):1028–1034, 2005.

[MdR07] Leonardo Michelon, Simone A. da Costa, and Leila Ribeiro. Formal specification and verification of real-time systems using graph grammars. *Journal of the Brazilian Computer Society*, 13(4):51–68, 2007.

[Mes01] Franz Mesch. Strukturen zur selbstüberwachung von messsystemen. *Automatisierungstechnische Praxis*, 43(8):2–7, 2001.

[MLK12] Tanja Mayerhofer, Philip Langer, and Gerti Kappel. A runtime model for fuml. In *Proceedings of the 7th International Workshop on Models@run.time (MRT 2012)*, 2012. Vortrag: 7th International Workshop on Models@run.time (MRT 2012), Innsbruck; 2012-10-02.

[MPW10] Nidhal Mahmud, Yiannis Papadopoulos, and Martin Walker. A translation of state machines to temporal fault trees. *Dependable Systems and Networks Workshops*, 0:45–51, 2010.

[MR12] Norma Montealegre and Franz Rammig. Agent-based modeling and simulation of artificial immune systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2012 15th IEEE International Symposium on*, pages 212–219, Shenzhen, China, 2012.

[MS02] J. Magott and P. Skrobanek. Method of time petri net analysis for analysis of fault trees with time dependencies. *Computers and Digital Techniques, IEE Proceedings* -, 149(6):257 – 271, nov 2002.

[MSKC04] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56 – 64, july 2004.

[MWP11] N. Mahmud, M. Walker, and Y. Papadopoulos. Compositional synthesis of temporal fault trees from state machines. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 429 –435, aug. 2011.

[NFPB12] Viet Hoa Nguyen, François Fouquet, Noël Plouzeau, and Olivier Barais. A Process for Continuous Validation of Self-Adapting Component Based Systems. In *7th International Workshop on Models@run.time of the MODELS 2012 Conference.*, Innsbruck, Austria, 2012.

[Nig09] Maik Niggemann. Risikoanalyse für die rekonfiguration selbstoptimierender mechatronischer systeme. Bachelor's thesis, University of Paderborn, Department of Computer Science, Paderborn, Germany, May 2009.

[NSS+11] Florian Nafz, Hella Seebach, Jan-Philipp Steghöfer, Gerrit Anders, and Wolfgang Reif. Constraining self-organisation through corridors of correct behaviour: The restore invariant approach. In Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer, editors, *Organic Computing — A Paradigm Shift for Complex Systems*, volume 1 of *Autonomic Systems*, pages 79–93. Springer Basel, 2011.

[ÖM07] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[Pal08] Sanjay Kumar Pal. 21st century information technology revolution. *Ubiquity*, 2008(June):9:3–9:3, June 2008.

[PR92] Irith Pomeranz and Sudhakar M. Reddy. On the generation of small dictionaries for fault location. In *Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, ICCAD '92, pages 272–279, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[PWP+11] Yiannis Papadopoulos, Martin Walker, David Parker, Erich Rüde, Rainer Hamann, Andreas Uhlig, Uwe Grätz, and Rune Lien. Engineering failure analysis and design optimisation with hip-hops. *Engineering Failure Analysis*, 18(2):590 – 608, 2011. <ce:title>The Fourth International Conference on Engineering Failure Analysis Part 1</ce:title>.

[Rau01] A. Rauzy. Mathematical foundations of minimal cutsets. *Reliability, IEEE Transactions on*, 50(4):389 –396, dec 2001.

[Rau02] Antoine Rauzy. Mode automata and their compilation into fault trees. *Reliability Engineering and System Safety*, 78(1):1 – 12, 2002.

[Rau03] Antoine Rauzy. A new methodology to handle boolean models with loops. *IEEE Transactions on Reliability*, 52(1):96–105, 2003.

[RD97] Antoine Rauzy and Yves Dutuit. Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within Aralia. *Reliability Engineering & System Safety*, 58(2):127–144, November 1997.

[RDV09] J. E. Rivera, F. Duran, and A. Vallecillo. A graphical approach for modeling time-dependent behavior of DSLs. *Visual Languages - Human Centric Computing*, pages 51–55, 2009.

[Rea11] B.C. Readler. *Verilog by Example: A Concise Introduction for FPGA Design*. Full Arc Press, 2011.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artifical Intelligence*, 32(1):57 – 95, 1987.

[Ren07] Arend Rensink. Isomorphism checking in groove. In Albert Zündorf and Dániel Varró, editors, *Graph-Based Tools (GraBaTs), Natal, Brazil*, volume 1 of *Electronic Communications of the EASST*. European Association of Software Science and Technology, September 2007.

[Reu90] Christophe Reutenauer. *The mathematics of Petri nets*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[RFP93] Paul G. Ryan, W. Kent Fuchs, and Irith Pomeranz. Fault dictionary compression and equivalence class computation for sequential circuits. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, ICCAD '93, pages 508–511, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[RS12] Martin Riedl and Markus Siegle. A LAnguage for REconfigurable dependable Systems: Semantics & Dependability Model Transformation. In *Proc. of the 6th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECOS'12)*, eWiC, pages 78–89. British Computer Society, August 2012.

[RSV13] Franz Rammig, Katharina Stahl, and Gavin Vaz. A framework for enhancing dependability in self-x systems by artificial immune systems. In *Proceedings of the Fourth IEEE Workshop on Self-Organizing Real-time Systems*, 2013. accepted.

[Rud84] Richard L. Rudell. Multiple-Value Logic Minimization for PLA Synthesis. Technical Report M86/65, University of California at Berkeley, USA, June 1984.

[Rus08] John Rushby. Runtime verification. In Martin Leucker, editor, *Runtime Certification*, pages 21–35. Springer-Verlag, Berlin, Heidelberg, 2008.

[SBT11] Daniel Schneider, Martin Becker, and Mario Trapp. Approaching runtime trust assurance in open adaptive systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 196–201, New York, NY, USA, 2011. ACM.

[Sed08] Milos Seda. Heuristic Set-Covering-Based Postprocessing for Improving the Quine-McCluskey Method. *International Journal of Computational Intelligence (IJCI)*, 4(2):139–143, 2008.

[SFP02] Silvio Simani, Cesare Fantuzzi, and Ron J. Patton. *Model-based Fault Diagnosis in Dynamic Systems Using Identification Techniques*. Springer Berlin / Heidelberg, 2002.

[Sha02] Mary Shaw. "self-healing": softening precision to avoid brittleness: position paper for woss '02: workshop on self-healing systems. In

Proceedings of the first workshop on Self-healing systems, WOSS '02, pages 111–114, New York, NY, USA, 2002. ACM.

[SHS11] Julian Suck, Christian Heinzemann, and Wilhelm Schäfer. Formalizing model checking on timed graph transformation systems. Technical Report tr-ri-11-316, Heinz Nixdorf Institute, University of Paderborn, September 2011.

[SRA04] Koushik Sen, Grigore Roşu, and Gul Agha. Online efficient predictive safety analysis of multithreaded programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 123–138. Springer Berlin / Heidelberg, 2004.

[Sto96] Neil Storey. *Safety Critical Computer Systems.* Addision Wesley, 1996.

[SW07] Wilhelm Schäfer and Heike Wehrheim. The challenges of building advanced mechatronic systems. In *FOSE '07: 2007 Future of Software Engineering*, pages 72–84. IEEE Computer Society, 2007.

[SWZ95] Andy Schürr, Andreas J. Winter, and Albert Zündorf. Graph grammar engineering with progres. In *Proceedings of the 5th European Software Engineering Conference*, pages 219–234, London, UK, 1995. Springer-Verlag.

[Tec08] Avago Technologies. Afct-57j5apz, afct-57j5apz-xxx reliability data sheet, 2008.

[TG98] Andrew S. Tanenbaum and James R. Goodman. *Structured Computer Organization.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 1998.

[TGS06] M. Tichy, H. Giese, and A. Seibel. Story diagrams in real-time software. In *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany*, 2006.

[THHO08] Matthias Tichy, Stefan Henkler, Jörg Holtmann, and Simon Oberthür. Component story diagrams: A transformation language for component structures in mechatronic systems. In *Postproc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany.* HNI Verlagsschriftenreihe, 2008.

[THMvD08] Matthias Tichy, Stefan Henkler, Matthias Meyer, and Markus von Detten. Safety of component-based systems: analysis and improvement using fujaba4eclipse. In *Companion of the 30th international conference on Software engineering*, ICSE Companion '08, pages 973–974, New York, NY, USA, 2008. ACM.

[TvS08] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall International, 2nd rev. ed. edition, 2008.

[VDI04] VDI. *VDI 2206: Entwicklungsmethodik für mechatronische Systeme*. Verein Deutscher Ingenieure, 2004.

[VG10] Thomas Vogel and Holger Giese. Adaptation and abstract runtime models. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 39–48, New York, NY, USA, 2010. ACM.

[VGc12] Richard Vaughan, Brian Gerkey, and contributors. Player/stage documentation, 2012.

[VGRH81] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. Fault tree handbook - nureg-0492209. Technical report, U.S. Nuclear Regulatory Commission, 1981.

[vOQ55] William van Orman Quine. A Way to Simplify Truth Functions. *The American Mathematical Monthly*, 62, 1955.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[WGR+09] Max Walter, Alexander Gouberman, Martin Riedl, Johann Schuster, and Markus Siegle. Lares — a novel approach for describing system reconfigurability in dependability models of fault-tolerant systems. In *Proc. of the European Safety and Reliability Conference (ESREL 2009)*, pages 153 – 160. Taylor & Francis Ltd., 2009.

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10:115–152, 5 1995.

[Zim08] Armin Zimmermann. *Stochastic Discrete Event Systems: Modeling, Evaluation, Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2008. Chapter 7.

[Zim10] Armin Zimmermann. Dependability evaluation of complex systems with timenet. In *Proceedings of the First Workshop on DYnamic Aspects in DEpendability Models for Fault-Tolerant Systems*, DYADEM-FTS '10, pages 33–34, New York, NY, USA, 2010. ACM.

List of Definitions

List of Figures

List of Tables